The University of Alberta

TIME REVISITED

by

© Stephanie A. Miller

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-42716-7

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Stephanie A. Miller

TITLE OF THESIS: Time Revisited

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1988

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) Stephanie Miller

Permanent Address:
#37 Roseburn Estates
52148 - Range Road 231
Sherwood Park, Alberta
Canada T8B 1A6

Dated 1 March 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Time Revisited** submitted by **Stephanie A. Miller** in partial fulfillment of the requirements for the degree of **Master of Science.**

L L Schulist

Supervisor

R Boerlel

Jia-Dwei You

Lois M. Stanford

Date March 1, 1988

ABSTRACT

Theorem provers are prone to combinatorial explosions, especially when dealing with certain kinds of inference. To compensate for this, special purpose inference methods have been developed that are more efficient within their subdomain than any general method. By exploiting properties of those domains, such mechanisms can shorten chains of reasoning with types, temporal relations, colors, arithmetic relations, and sets, to name a few.

The problem investigated here is one of how best to use these efficient, but limited methods in a more general environment. Although much research has been done on this problem, most of the resulting systems either restrict what they can represent and reason with, limit the types of special mechanisms that can be used, or are difficult to extend with other specialists.

A uniform interface to the specialists is developed which allows them to assist a resolution-based general theorem prover in function evaluation, literal evaluation, and generalized resolving and factoring. The special mechanism used as the main test bed for this research was a temporal specialist. This dissertation takes an efficient temporal reasoner and extends its inferential capabilities to handle both strict and non-strict ordering. Further enhancements enabled the temporal specialist to do generalized resolving and factoring, as well as function and literal evaluations. The resulting temporal specialist is incorporated into the system using the specialist interface.

The emphasis throughout is on applied rather than theoretical issues. Experience with the interface confirms that new specialists can be added with relative ease. This is because the specialists interact with the general theorem prover in a small, fixed set of ways. Each new specialist was found to make possible fast proofs of questions previously beyond the scope of the theorem prover. The combined system can, in a few steps, do some proofs that would normally require numerous steps. Examples from the fully operational hybrid system are included.

Acknowledgements

First, thanks are due my supervisor, Dr. Len Schubert, who gave me the inspiration for this work. I'd also like to thank the members of my committee: Dr. Lois M. Stanford, Dr. Randy Goebel, and Dr. Jia You, for taking the time to read my thesis.

Financial assistance from Boeing, the Province of Alberta, and the Department of Computing Science is gratefully acknowledged.

I'd like to express my appreciation to the staff members of the Department of Computing Science for the various forms of technical and moral support they gave me. And last, but certainly not least, thanks to Jonathan Schaeffer, without whose continual nagging this thesis might never have been finished.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Specialists	4
Chapter 2: Use of Specialists	8
2.1. Hybrid Reasoners	8
2.1.1. Restricted Language Hybrids	9
2.1.2. Theorem-Prover-Based Hybrids	20
2.2. Blackboard Approaches	23
Chapter 3: The Temporal Specialist	25
3.1. The Taugher and Schubert Model	30
3.2. Changes to the Original Temporal Specialist	33
3.2.1. Strict and Non-Strict Relations	34
3.2.2. Predicates and Argument Patterns	38
3.2.3. Duration Determination	41
3.3. Use of the Temporal Specialist	42
3.3.1. Function Evaluation	44
3.3.2. Literal Evaluation	45
3.3.3. Generalized Resolution and Factoring	46
Chapter 4: The Specialist Interface	54
4.1. Changes Required to the General System[12]	55
4.1.1. Sorts	56
4.1.2. Quoted Expressions	58
4.1.3. Unification	58
4.1.4. Interface	59
4.1.5. Resolution	60
4.2. Specialist Requirements	61
4.3. The Specialist Interface	62
4.3.1. Function Evaluation	63
4.3.2. Literal Entry and Evaluation	64
4.3.3. Generalized Resolution and Factoring	65
4.4. Communication between Specialists	66
4.4.1. Additional Specialists	67
4.4.2. Immediate Evaluation	69
4.4.3. Delayed Communication	69
Chapter 5: Experimental Results	73
5.1. Examples	73

5.2. Positive Results 77
5.3. Negative Results 78
5.4. Comparison with Other Systems 79
Chapter 6: Conclusions and Further Work 80
6.1. Conclusions 80
6.2. Extensions and Further Work 81
6.2.1. Closure 82
6.2.2. Other Theorem Provers 84
6.2.3. Specialists 84
6.2.4. Effort Levels 86
6.2.5. Parallelization 86
References 88
Appendix 1: Analysis of the Number of Cross Chain Links 90
Appendix 2: Closure Algorithm for Taugher and Schubert Model 91
Appendix 3: Example Concept Hierarchy 93
Appendix 4: Example Topic Hierarchy 96
Appendix 5: General Knowledge Used in Examples 100
Appendix 6: Little Red Riding Hood Knowledge 103
Appendix 7: Sample Run of ECoNet 106

List of Figures

Figure	Page
2.1 Architecture of CAKE[1]	10
2.2 Architecture of KL-TWO	12
2.3 The architecture of Mecho	13
2.4 Architecture of the Nelson-Oppen System	16
2.5 Architecture of Shostak's system	19
2.6 Architecture of Krypton	21
3.1 Example Timegraph and Metagraph	30
3.2 Algorithm to Determine Duration	42
4.1 Architecture of ECoNet	54
5.1 Example of ECoNet in operation[17]	74
5.2 Example showing Communication between Specialists[18]	76

Chapter 1

Introduction

A common requirement in many areas of Artificial Intelligence is to control the combinatorial explosion associated with searching, whether in a theorem prover, problem solver, planner, or game playing program. Two main approaches have been tried: better search algorithms, and restricting the domain so that the search space is not so large. Ad hoc methods and heuristics fall under the first category. Ad hoc methods work well in their limited domains, but are not easily extensible nor necessarily complete. Heuristics for pruning search spaces reduce the search space, but at the expense of possibly missing something that could, for example, lead to a winning move in a game. Restricting the domain leads to systems that can only solve uninteresting problems.

However, for some well known areas, special purpose inference methods have been developed that reason faster than any general method can by exploiting the properties of their own small domain. These *specialists* may use completely different representations and methods to achieve their performance. For example, a type specialist can use a type hierarchy to short-cut the chain of reasoning involved in determining that a wolf is a living-thing (the sequence of inferences *wolf -> warm-blooded-quadruped -> larger-animal -> animal -> creature -> living-thing* can be reduced to *wolf -> living-thing*). Similarly, a time specialist would compress reasoning about transitive temporal orderings, such as inferring that event1 is before event4, given that event1 is before event2, event2 is before event3, and event3 is before event4. Other specialists could deal with colors, arithmetic relationships, sets, spatial relationships, and so on. The question then arises as to how to use these efficient, but limited methods in a more general environment. This is the problem central to this dissertation. The most promising solution would integrate a general mechanism with the specialized mechanisms, using the specialists when appropriate, and the general method when no specialists apply. This would give us a system with a wider domain than all the specialists combined, avoiding much of the combinatorial explosion

usually associated with a large domain. Such a system would be extremely useful for low level inferencing in natural language understanding, program verification, and other applications.

Our emphasis here is on the development of practical algorithms, rather than formal theoretical development. A design assumption important to our system is that the general system can fill in any inference gaps in the specialists, even if inefficiently. This means that there is no requirement that a specialist be complete, although they must make sound inferences.

Several approaches for integrating different inference methods have been suggested, among them blackboard approaches [EHL80], use of equality information [NeO79, Sho84], hybrid architectures consisting of a theorem prover and special methods [BFL83, SPT87], restricted language hybrids [Ric85, Vil85], and so on (see Chapter 2). Blackboard systems are usually complex and slow, and their flexibility is overkill for the problem we are looking at here. Restricted language hybrids are restricted to handle only efficiently decidable subsets of predicate logic, so it is difficult to tell how much they contribute towards our goal. The Shostak and Nelson-Oppen systems apply only to certain kinds of theories - unquantified equational theories - and so still give us a limited domain. Some of the hybrid approaches look promising, but do not incorporate enough specialists to tell if their approach is general enough for our purposes.

This thesis investigates a hybrid approach consisting of a resolution based theorem prover which operates on a semantic net, with a temporal inference specialist (based on the discussion in [SPT87]). Although hybrid approaches have been tried for this problem before, most use disjoint specialists, and communication among specialists remains an issue. A specialist may break its problem down into subproblems, which may require the help of another specialist (or even the general method). The specialist should not have to know which other specialists are available (if any) to solve its sub-problems and how to invoke them. The specialist interface should make the use of the specialists invisible to each other. This makes it easy to add new methods without worrying about changing all the interactions among the existing ones. Another place where

communication is required is in term evaluation - if one specialist gets a term that it cannot evaluate (for example, a function), another specialist may be able to.

The temporal specialist used here is the test bed for most of the interface research. It was chosen for several reasons:

1. it involves all the possible interfaces between the general and special methods,
2. it requires communication with other specialists (in particular a number specialist),
3. the type of inference it does is well understood, and
4. an efficient algorithm for implementing it was available.

Although the temporal specialist algorithm used was efficient, it was not complete in several respects, notably in that it could only "handle" a subset of the possible simple temporal relations among points. Some enhancements were done to this specialist to compensate, as well as to extend the capabilities and expressiveness of the specialist. Generalized resolution and factoring were also implemented. Generalized resolution does not require literals being resolved to have identical predicates of opposite "sign"; instead, the two literals may be complementary in a more general sense, as in the case of the "taxonomic" pairs of literals *[d dog]* against *[d cat]*, or *[d dog]* against *~[x animal]*. In these examples we can directly infer the null clause. Similarly, generalized factoring may involve non-identical predicates such as *[d dog]*, *[x animal]*, which have generalized factor *[d animal]*. Additional examples, involving temporal literals, will be seen below. Generalized resolution was discussed in [PaS82] (where it was termed "extended resolution") and in [SPT83]. Both generalized resolving and generalized factoring are special cases of Stickel's *partial theory resolution* [Sti83].

The research for this thesis originated with some enhancements to the temporal specialist, especially in how it could be extended and used in a more general environment. Hence the title "Time Revisited". The combined inference system developed here is intended for low level inferencing in a natural language understanding system (**ECoSystem** [HaS86]) under development at the University of Alberta. This portion of **ECoSystem** is called **ECoNet**. It accepts

assertions in the form of first order predicate logic propositions, and answers questions phrased in the same form. The system is implemented in Lucid Common Lisp. Several specialists are incorporated, including the temporal specialist already mentioned, an arithmetic/number specialist, a simple color specialist, a type hierarchy specialist, and a set/list specialist.

The rest of this chapter further introduces specialists. Chapter 2 reviews some previous work on combining different representations and methods. Chapter 3 introduces temporal inference and discusses enhancements to the temporal specialist. Chapter 4 describes the specialist interface design for this thesis, including the modifications required to both the theorem prover and the specialist. It also discusses how communication among the specialists was achieved. Most work on combining specialists has been done with disjoint specialists, leaving an open area in dealing with overlapping specialists. To test this communication, two more interacting specialists were implemented: a number specialist, and a set/list specialist. The domains of these specialists as well as their interactions with the temporal specialist are also discussed. Chapter 5 discusses the results, positive and negative, of this approach, and compares this approach to the other systems mentioned in Chapter 3. And finally, Chapter 6 contains conclusions, and a discussion about possible extensions and further work.

1.1. Specialists

Before going any further, we must first decide what is meant by *specialists*, or special purpose inference mechanisms, and what can be gained by using these methods. Special inference methods take advantage of special properties of the predicates, terms and functions in the domain they work with to develop efficient representations and methods for reasoning in that domain. The inference done by the specialists in their own domains is much faster and more efficient than is possible for a general inference mechanism. For example, the temporal specialist uses a graph structure to represent entities and relations passed to it, and uses graph searching techniques that have been studied and improved on for many years to do its reasoning. Because of the transitive

nature of temporal relations, making inferences with them is computationally expensive in most general methods.

Efficiency alone is motivation enough for using specialists, but there are other reasons as well. Specialists can allow several forms of input language, giving more flexibility of expression to the user. However, we cannot just allow expressions outside the syntax of the general inference system to be used - this would extend the input language in a haphazard way, leading to unpredictable "holes" in what can and cannot be handled. The work on specialists here indicated a need for an extended input syntax, but to the general logical syntax and inference methods as well (e.g. for numbers, equality, sets, and quoted expressions). For practical reasons, most of these have not yet been implemented into the general reasoner.

Using specialists also gives a more modular approach to a system that does many kinds of inference - each specialist then only manages a small portion of the whole problem, leading to simpler and more efficient algorithms. Compared to a general purpose theorem prover that handles all domains, not only can the specialists increase efficiency, they can provide a cleaner and more elegant implementation of the whole system.

There have been many specialists developed for different domains. Some are for things that people do "without thinking" such as inferring simple color, time, part-of and taxonomic relationships, etc [Pap82, PaS82, SPT83, SPT87]. People do this "trivial" inference without even realizing it, while a long string of axioms and therefore inferences is required for most theorem provers. These specialists are interesting in that one must first discover a method for mimicing the types of inference people do. Inferences made by such specialists have to be sanctioned by an axiomatic theory compatible with our intuitions about their respective domains - time, taxonomies, sets, colors, etc.

Others are for well defined domains such as integer or real arithmetic, lisp functions, equality, and so on [NeO79, Ric85, Sho84]. These domains usually have a logical or mathematical

basis and axioms one can look up in a textbook. These will be called "mathematical" specialists.

The temporal specialist used throughout this thesis is a "without thinking" type, although some of the "mathematical" variety are required to assist it (especially a number specialist to be described later).

Still other special methods are for specific applications that don't necessarily have a rigidly defined or even well understood background. For example, in the HEARSAY-II speech-understanding system [EHL80], some of the information routines use probabilistic methods to solve their subproblems (including the grammar used to determine input sentence structure). These are usually called *knowledge sources*, which is a more general term than *specialist*. A *specialist* can be considered to be a *knowledge source* with some degree of proficiency.

Loosely defined, a specialist is an inference mechanism (or knowledge source) that operates on a specific and well-defined domain. Within that domain, it may employ some non-standard inference methods in order to solve a problem more efficiently. This "definition" is slightly different from *decision procedure* (an algorithm for determining whether a formula is valid in a logical theory, [NeO79]) and *theory* (a set of related axioms, [Sti83]), in that efficiency is a requirement. The specialists involved in the system implemented here are for theories which can be expressed as a set of first order axioms. In this case, the specialist can be viewed as a "compiled" version of the axioms that is more efficient than applying a general inference method to the individual axioms.

Schubert et al. [SPT87] suggest several ways for a specialist to accelerate a theorem prover using derived rules of inference, including literal evaluation and generalized resolution and factoring. They discuss the relationship to Stickel's theory resolution in detail. In addition, a specialist can evaluate functional terms to simplify literals. The temporal specialist bypasses the normal proof procedure for the operations it handles, and can cut out numerous proof steps that would otherwise be required. Literal evaluation uses a specialist's special representation to simplify

assertions, and resolvents generated by the theorem prover. Function evaluation simplifies a term by evaluating it (for example, *(max-of n1)* can be simplified to a number, say *3* by the number specialist). Generalized resolving and factoring quickly determine incompatibility or subordination of one literal by another. This allows resolution and factoring to be done where they usually cannot. For example, the temporal specialist can factor *[~[x during a] or [x during b]]* to *[a during b]* (if we have *[a during b]* represented in the timegraph) even though the literals have different signs. In addition, even though we may not have incompatibility, a specialist can determine the conditions for incompatibility and return these as a *residue*. A *residue* (from Stickel's partial theory resolution), is a literal or a set of literals whose negation would make the two literals being resolved incompatible (resolvable in one or more steps to the null clause). For example, if we resolve *[a before b]* against *[a after b]*, we get a residue of *[a equal b]*. If that residue is later determined to be false, we have the null clause. Further details of this will be discussed in Section 4.1.5. Similarly, if we know *[a before-1 b]* (*before-1* means strictly before), the temporal specialist can resolve *[x before a]* with *[x after b]* to the null clause, although the predicates are not identical, and the signs are the same.

The interface designed and implemented here is able to handle any specialist with a well defined theory (clearly specified domain and legal inferences in that domain) that can be used to accelerate resolution either through literal evaluation or generalized resolution or factoring. Although much of it relates to the resolution based theorem prover it is enhancing, in principle, the basic ideas should be applicable to other types of theorem provers also, as well as general representations other than semantic nets. This will be discussed later.

Aside from the discussion on the temporal specialist, we will concentrate mainly on how a specialist may be used to accelerate general inferencing, rather than the methods and representations each specialist uses.

Chapter 2

Use of Specialists

As mentioned earlier, there are numerous ways to combine different representations and inference methods. This chapter describes some of these approaches. Several issues need to be considered in comparing these systems, some of which are pointed out in [BrR85]. There is no clear-cut division into groups, as the issues involved are not independent. The breakdown of the systems in this chapter is somewhat arbitrary and is certainly not the only one possible. This categorization just reduces the number of systems each system must be compared against. Instead we compare some general features of the groups and then compare the systems within each group.

2.1. Hybrid Reasoners

A hybrid architecture is one that uses multiple representations and/or methods to do its reasoning, rather than attempting the entire task of knowledge representation and inferencing with a single, uniform component. The subcategorization into *Restricted Language Hybrids* and *Theorem Prover based Hybrids* for this section was suggested by Vilain [Vil85]. Another classification is possible for these as well - *essential* and *non-essential* hybrids [BGL85]. A non-essential system may represent facts in more than one format since another method is available. A specialist in such a system is considered non-essential, although it certainly may have significant advantages over the general representation (like making inference more efficient). Essential systems do not have this duplication.

2.1.1. Restricted Language Hybrids

A restricted language reasoner starts with a restricted version of logic that can represent only a subset of the things one might want to use in a system. However, inference in that restricted language is usually efficient. Many of these systems are based on a reasoning package called RUP (Reasoning Utility Package - McAllister), which is a subset of first order logic. Other possibilities for restricted languages to build systems on are forms of relevance logic (for example, [Pat85]).

As the core of the system is limited in what it can represent or reason about, specialists are required both to give it more expressiveness, and to reason with the additional representations. These systems would appear to be "essential" hybrids then, as the specialists provide another representation that expresses things the core cannot. It is still possible to get duplication of facts in several representations though (as CAKE does, which we will see shortly). Perhaps it would be better if each specialist of a system were classified as essential or non-essential (rather than the system itself) based on whether the core of the system or main driver can express the facts the specialist can.

One example of a restricted language hybrid is CAKE [Ric82, Ric85] used for program verification. It consists of specialists hierarchically arranged on top of a RUP-based core. There are two levels of language - a plan calculus and predicate calculus. This system implements the Plan Calculus level on top of the Predicate Calculus level, so that the planning language coexists with its translation in the predicate calculus. Explicit connections exist between the two levels, allowing changes in one level to be incrementally translated to the other.

In addition to the level division, the system is also divided into eight layers, each with associated routines and data structures. The layers correspond to levels of data abstraction. Each layer adds an increment of functionality to the system. Three of these layers are in the plan level,

although only the lowest one, the Plan Calculus layer, is implemented. The other five layers are in the predicate calculus level. Interactions between layers are accomplished by subroutine calls. A layer may call any subroutine in a layer beneath it, but none higher (the layers are not arranged strictly hierarchically in the interests of optimization).

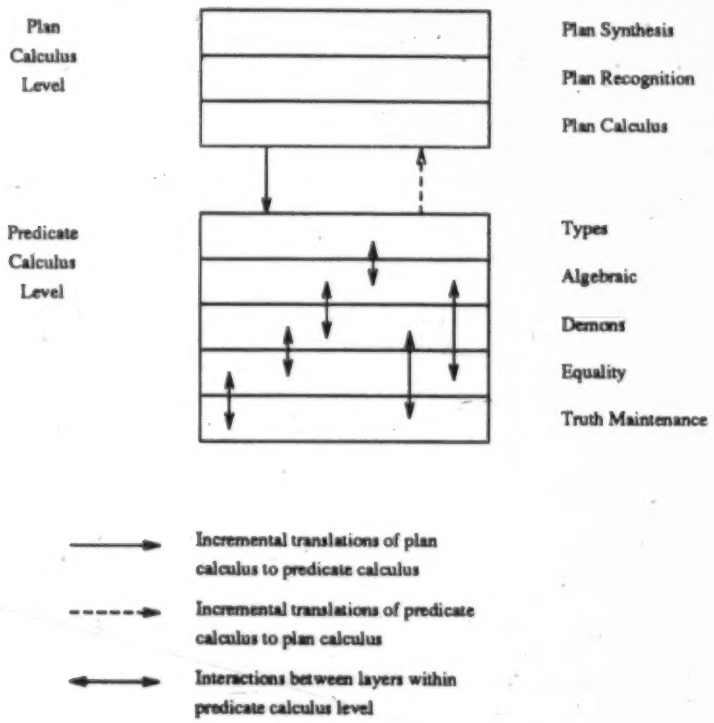

Figure 2.1 Architecture of CAKE[1]

Given a plan consisting of a set of steps, the system can determine whether the plan is a correct implementation of a particular operation with associated input and output. It does this by assuming the preconditions of that operation and trying to prove its postconditions, using the

[1] The system diagrams in this section are intended to show the important differences between the architectures and may not necessarily accurately express details of the systems they represent.

plan. It tries to prove the preconditions of the first step in the plan, and if they are satisfied, it asserts that operation's post-conditions. Then it proves the preconditions of the next step, and so on, until finally it proves the post-conditions of the operation the plan is for. For any of the proofs involved, the specialists assist. For example, if we are trying to prove that a certain sequence of operations will calculate the square root of an integer, one step will take the integer input, and have a real number output. A later step may require as its precondition that the input be of type number, at which point the type layer (specialist) will be invoked to determine that an object of type real is also of type number. The algebraic layer may be involved as some of the operations are commutative. The equality layer will keep track of and use, among others, the equality between the input to the square root operation and input to the plan, and the equality between outputs of the operation and the plan.

This system makes two main contributions - it allows two different views of the same object (a plan), and it involves the use of several different inference mechanisms to solve problems. A drawback to this particular approach is that each specialist must know about any other specialist that can help it, and how to invoke it. The method used to interface the specialists appears ad hoc. As Rich notes [Ric85], this leads to possible computational explosions when new levels or layers are added, as the interactions are so highly tuned.

Another of these hybrids, KL-TWO [Vil85] also has a RUP-based core (called PENNI), extended with a limited quantification/terminological component (called NIKL). PENNI is a quantifier-free version of predicate calculus with equality, and deals only with grounded propositions. NIKL contains complex, intertwined term descriptions, which correspond to one and two place predicates. It classifies new terms and caches subsumption information about them for later use. NIKL basically deals with terminological (type) information, and the only quantification involved is that each NIKL sentence can be viewed as being universally quantified. For example, if NIKL contains information to the effect that an animal is a type of creature, that can be viewed

as expressing the quantified proposition

(1) [A x [x animal] => [x creature]]

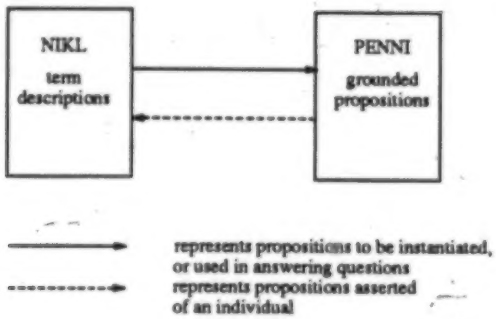

Figure 2.2 Architecture of KL-TWO

When a proposition about an individual is received by PENNI, it is forwarded to NIKL. A forward reasoner then determines which subset of NIKL sentences apply, and may send one or more back to PENNI for instantiation. NIKL may then create new terms and relate those to existing terms. For example, if we also had information in NIKL that defines a carnivore as a type of animal that eats other animals, and we now assert the propositions[2]

[WOLF animal]
[LRRH animal]
[WOLF eat LRRH].

NIKL would then construct an abstraction term for *WOLF*, and determine subsumption relations, one of which would be *(1)*. The abstracted term would canonicalize to *carnivore*, and NIKL would assert to PENNI the ground clause

[[[WOLF animal] & [LRRH animal] & [WOLF eat LRRH]] => [WOLF carnivore]].

(an instantiation of one side of the equivalence relation defining *carnivore*). During a query, the cached subsumption information is used by a backward reasoner to determine which sentences to

[2] LRRH in these examples means Little Red Riding Hood. The test examples for most of this work was the story of Little Red Riding Hood.

apply. For example, given the above assertions, if later the question *?[WOLF creature]* is asked, *(1)* will be instantiated for the individual *WOLF* and given to PENNI to use in answering the question.

The strength of this system lies in the complex term descriptions and classification handled by NIKL. The specialized language and inference mechanism make such descriptions easy to enter and efficient to reason with, where in a general logic-based representation this could be clumsy and computationally expensive. However, the entire system is unable to stand on its own and must be extended for any application. As NIKL is mainly a terminological component, it handles only universal quantification. Only two inference methods are combined here, and it is not clear if the approach is general enough to add another. Vilain [Vil85] claims that the system is efficient although it relies on some incomplete algorithms.

Bundy et al. [BBM82] use several domain independent specialists in a mechanics problem solver **Mecho**. This system solves mechanics problems stated in English. This requires inference for interpreting the English statements, forming a mathematical model to solve the problem, and solving the resulting simultaneous equations. Their approach is to use algebraic properties of predicates (such as symmetry, transitivity, etc) and develop specialized mechanisms for these properties. This approach allows them to take advantage of these properties for any domain they decide to use them for, without having to develop new specialists. However, they do not discuss their basis for distinguishing a domain-dependent specialist from a domain-independent specialist. In fact, there is no sharp distinction. If some specific set of algebraic properties (of relations and functions) is found only in some narrow domain, then special methods designed to exploit those properties could be called "domain-dependent". If the properties are found in a lot of different domains, the special methods are "domain-independent". From that point of view, the special methods added to ECoNet (types and time especially) are domain-independent since types and times are important in a lot of different domains (natural language, planning, etc).

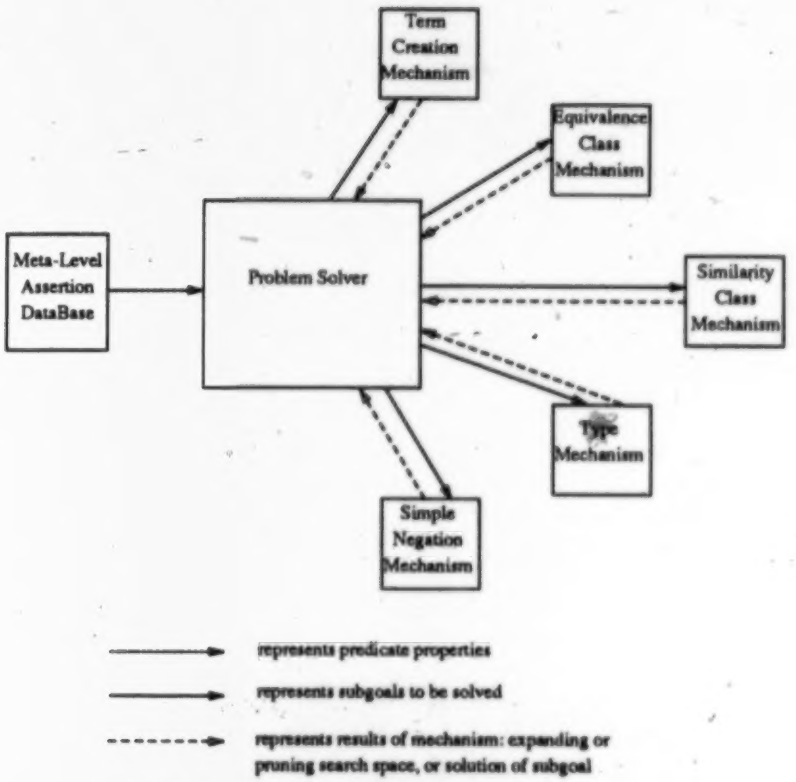
Term Creation Mechanism
Equivalence Class Mechanism
Meta-Level Assertion DataBase
Problem Solver
Similarity Class Mechanism
Type Mechanism
Simple Negation Mechanism
represents predicate properties
represents subgoals to be solved
represents results of mechanism: expanding or pruning search space, or solution of subgoal

Figure 2.3 The architecture of Mecho

The specialists in Mecho include equivalence and similarity mechanisms, a unary predicate (type) mechanism, as well as some special methods to control expansion of the search space (through term creation), and to prune the search space (through simple negation checking). These mechanisms are implemented mainly by introducing new inference rules. Although the equivalence and similarity class mechanisms are discussed in terms of tree structures, they are actually implemented by replacing the conventional axiomatization of such properties with a less explosive one. Then for each predicate with the appropriate property, the new axioms are instantiated for it, and used when solving problems with clauses involving that predicate. For example, the predicate *sameplace* has the property of *quasi-equality* (similar to equality but with an extra

argument - usually for time). The following axioms would result from the instantiation for *sameplace*:

Rootof-sameplace(x, root, t) & Rootof-sameplace(y, root, t) => sameplace(x, y, t)
This is the main axiom, stating that two objects are in the same place at time *t* if they are in the same tree at that time.

Arc-sameplace(x, next, t) & Rootof-sameplace(next, root, t)
=> Rootof-sameplace(x, root, t)
If two objects are connected, they are in the same tree.

[~ E next Arc-sameplace(x, next, t) => Rootof-sameplace(x, x, t)
If an object is not connected to anything, it is the root of a tree.

Meta-level inference is used to allocate inference goals to the appropriate special mechanisms, by inferring properties of the goals from meta-level assertions of properties of their syntactic parts (predicates and arguments). For example, the predicate *sameplace* has the property *quasi-equality*, so a goal involving that predicate (like answering *?sameplace(LRRH, WOLF, wolf-eat-lrrh-episode)*) would be shipped off to the equivalence class mechanism. It is not clear whether the axioms involved in that mechanism are instantiated only when it is invoked to solve a problem, or whether they are instantiated as soon as the meta-level assertion about the predicate's property is made. A database called a *metabase* keeps the meta-level assertions about properties. Unfortunately, there is no indication of how efficient this meta-level inference is. For example, how long does it take to determine which specialist would be interested in a goal? If a goal has a predicate with properties x and y, how is the appropriate special mechanism determined using those properties?

A severe drawback to this approach is that the "special mechanisms" must be in the form of specialized logical inference rules, and cannot be special representations, such as graphs. This limits the types of special mechanisms that may be added, and prevents more efficient representations from being used. For example, the temporal specialist used in ECoNet uses a graph representation which not only efficiently handles the transitive properties of temporal relations, but also the cumulative nature of dates and durations. Finding an axiomatization which can

achieve the same efficiency, if there is one, could be a difficult search, and not warranted when we already have an efficient method (the graph).

Nelson and Oppen [NeO79] developed a system that simplifies expressions for use in a program verifier. Their system combines four different satisfiability procedures (algorithms for determining whether a set of literals is satisfiable in a logical theory) into a single decision procedure for the combination, which manages the predicates and functions of all of them. The decision procedures are:

R - real numbers under + and ≤
A - arrays under *store* and *select*
L - list structures under *car*, *cdr*, *cons*, and *atom*
E - equality with uninterpreted function symbols

The resulting procedure uses what they call *equality propagation*, where uninterpreted mixed terms in a literal are decomposed into several equalities that each involve only one theory. New variables are introduced to act as placeholders for the mixed terms. For example, in determining the satisfiability of $[y \leq x + car(cons(0,x))]$, which involves mixed numeric and list structure terms, we would need to introduce a variable, say $[g1 = car(cons(0,x))]$. This would simplify the original equation to $[y \leq x + g1]$, all of whose terms and predicates are interpretable within the number theory. Now the equality given for $g1$ is almost entirely related to list structure theory, except for the number 0, so that would be replaced by another variable, $[g2 = 0]$, and this equation would go to the number theory, leaving $[g1 = car(cons(g2, x))]$ for the list specialist. Each decision procedure computes entailed equalities between these variables, and propagates this information to the other specialists. For example, in the above example, the list specialist can determine that $[g1 = g2]$, and propagate this to the number specialist, which can then determine that $[y \leq x + g2]$, and consequently $[y \leq x]$. For disjunctions of equalities, case splitting is done, where each possible "case" in the disjunction is used, one at a time.

According to Nelson and Oppen, it is never necessary to propagate disequalities, nor equali-

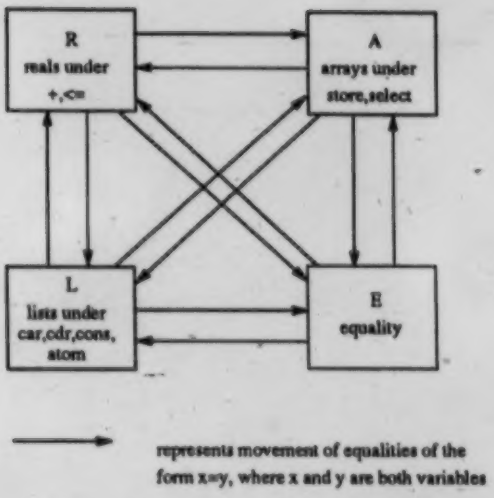

Figure 2.4 Architecture of the Nelson-Oppen System

ties other than between variables. If a disequality is noted by a specialist, the only time this will be of any use is in the presence of an equality relation between the two terms, in which case we have discovered that the equations are unsatisfiable. If such an equality is found, it will be propagated to all specialists, including the one that detected the disequality, so the inconsistency will still be detected. Equalities with terms other than variables are of no use as the other procedures may not understand the constants or functions. For example, propagating *[g2 = 0]* in the above example to the list specialist would be useless, as the non-variable part *0* is not recognized by the list specialist. Similarly, the functions *car* and *cons* are not recognized by the number specialist.

This architecture requires that each specialist know about all the other specialists, as communication is directed to all. This could be avoided by using a broadcast mechanism that each specialist could use. Even so, it shouldn't be neccessary to propagate the equality information to all specialists (as long as the subproblems are independent, which they are here). The information should only be propagated if there is some chance that if could be useful. In a small system with

few variables this won't make much difference, but in a more realistic system with hundreds or even thousands of such items, the communication must be controlled or the system will be too slow to be of any practical use. Also, this system allows only equality information to be communicated between specialists. As we will see in Chapter 4, there may be other useful kinds of information which can be communicated, such as the maximum of a number. Extension of the Nelson-Oppen architecture with additional specialists (with the required properties) would be straightforward, although cumbersome, because of the communication requirements.

Another problem with this system is that it is not clear how the specialists initially get their equations. Is there a high level decision procedure that selects the applicable specialist? Or are the equations propagated to all, like the equality information, and each specialist is then responsible for throwing out equations with terms and predicates not in its domain? Unlike Bundy's system, there do not appear to be any restrictions on how the actual special methods are implemented (special representations may be possible). However, the types of problems this system can handle appear to be limited to those that can be expressed by sets of equations. The specialists themselves must be for quantifier-free theories. This is not too restrictive as long as the main system can handle quantification, which this system cannot. Also, the system does not consider combinations of theories whose function and predicate symbols overlap (except = and ¬).

Shostak [Sho84] also developed a system that does simplification for a program verifier. It is similar to the Nelson-Oppen system in that information about equality is used for the simplification. However, it does not couple the separate decision theories, instead using a single uniform procedure that attempts to minimize the changes required to add each new specialist. In this system, there is no need for explicit computation of entailed equality information, or redundant representation of this information in the separate specialists.

The specialists must be equational theories closed under combination. (Stickel [Sti83]

defines an equational theory as one that can be expressed as a set of equalities or equivalences of a pair of literals). Several specialists are described as fitting into this class, and one can only assume that at least some of them are in Shostak's system. The specialists cover the convex theory of *car*, *cdr*, and *cons* (excluding *nil*), integer linear arithmetic, the Venn diagram monadic set theory (inclusion, union, intersection, and complement), and the unquantified first order theory of equality.

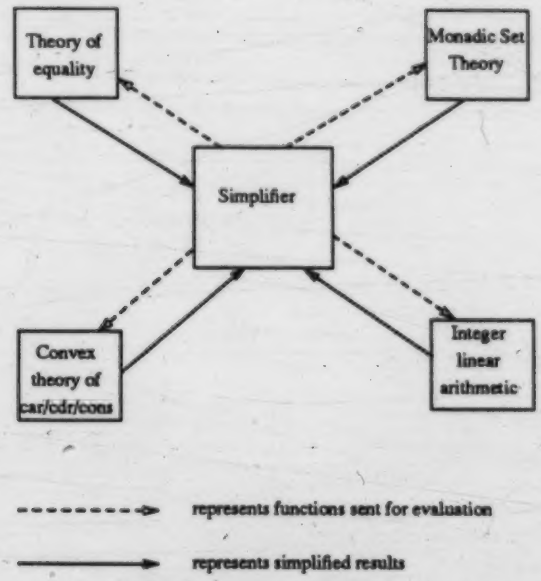

Figure 2.5 Architecture of Shostak's system

The process starts with a set of equalities (or disequalities) that it attempts to prove satisfiable or unsatisfiable. Terms from these equalities are evaluated (interpreted) incrementally by breaking them into subterms and simplifying those. For evaluation, the appropriate specialist for that function is applied, although Shostak does not indicate how this decision is made. If any of the equalities is determined to be false, the set of equalities is unsatifiable; otherwise it is satisfiable. For example, an equation

$$3 + x = car(cons((x + 2),y)) + 1$$

will involve number theory to simplify to

$$x = car(cons((x + 2),y)) - 2$$

and then list theory to get

$$x = x + 2 - 2$$

and finally number theory could determine satisifiability.

This system is similar to that of Nelson and Oppen in that it only handles problems which can be expressed in a set of equations, and uses equality information in combination with special methods to simplify the equations. It does not handle quantification either in the special methods or the main problem solver. However, unlike the previous system, this one appears to only handle functional symbols from each theory. Where the Nelson-Oppen system could also handle predicates such as ≤, this one does not appear to be able to. Many of the questions and comments from the Nelson-Oppen system apply to Shostak's as well.

2.1.2. Theorem-Prover-Based Hybrids

These hybrids use a first order logic theorem prover augmented by special methods that bypass the normal proof procedure and cut out some of the combinatorial search. They do not change the expressiveness of the system; they just accelerate inference and make it more efficient. In this sense, the specialists making up this type of system are "non-essential".

Stickel's theory resolution [Sti83, Sti85] gives a theoretical basis for using specialists in such a system. He has developed a set of complete procedures for incorporating theories into a resolution based theorem prover, which make it unnecessary to resolve directly on the axioms of the theory. Two types of theory resolution are available: *total*, which requires that the decision procedure (specialist) be capable of determining inconsistency of any set of clauses in the theory, and *partial*, which requires that the decision procedure be able to determine a set of substitutions and conditions for incompatibility of any pair of literals in the theory. As these can be undecid-

able in general, we must restrict any specialists which attempt them to the decidable subsets of their domains. For most specialists, partial theory resolution is easier to implement than total. Literal evaluation with result "false" (allowing elimination of that literal) can be considered narrow theory resolution with n=1, and generalized resolution can be considered the same with n=2 (from [SPT87]). Literal evaluation with result "true" (allowing elimination of the remaining literals) can be regarded as a form of theory-based subsumption elimination. In addition, Stickel's work can be extended to include "theory factorization". Theory factorization is similar to resolution in that the negation of the factor is incompatible with the other literal.

One example of such a system is **Krypton** [BFL83, BGL85]. Krypton has two components: a first order predicate calculus theorem prover ABox, and a taxonomic component (frame-based) TBox. The TBox here is essentially a subset of the NIKL component of KL-TWO, while the ABox handles a much richer (though still restricted) version of logic than KL-TWO's PENNI. The interactions between the components are different as well, although both systems look similar and started from similar roots. Krypton accepts two different languages and these are represented differently in the two components. Although this makes the system look like a Restricted Language Hybrid, the ABox can represent anything the TBox can, albeit not as naturally or efficiently. Thus there could be two representations for the same set of facts, although care has been taken to avoid this redundancy. Because the ABox is so expressive, Krypton fits into the theorem-prover-based hybrid group rather than the restricted hybrid calls.

The TBox is concerned with concepts and roles (term descriptions), and maintains subsumption and disjointness information about the terms. For example, it could represent the fact that a *carnivore* is an *animal* that eats other animals. Like KL-TWO, it has a caching scheme to save subsumption relations. The ABox handles a function-free subset of first order logic, and uses a connection graph resolution method based on Stickel's partial theory resolution. The predicates in the ABox are the terms in the TBox. The interface between the ABox and TBox

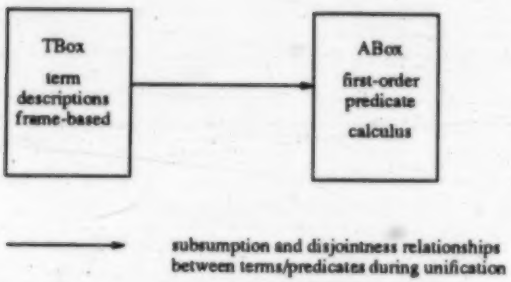

Figure 2.6 Architecture of Krypton

was achieved by augmenting the unification algorithm to incorporate the definitional relationships between predicates defined in the TBox. Extra links are added to the connection graph that are derived from the term definitions in the TBox, corresponding to subsumption relationships, for example. These are then used during the resolution attempt to answer a question. For example, given

[WOLF animal]
[LRRH animal]
[WOLF eat LRRH]

If we now ask *?[WOLF carnivore]*, the system would use the TBox to resolve the negation of this with the assertion that the wolf is an animal, giving a residue of

[E x [WOLF eat x] & [x animal]].

The negation of this residue then resolves against the other two clauses, giving the null clause, finishing the proof. Note that giving the same set of clauses to KL-TWO resulted in a slightly different sequence of events. KL-TWO would automatically make and store the inference that the wolf is a carnivore on assertion, whereas Krypton waits to make the inference until it is necessary to answer a question.

With only one specialist augmenting the system, it is difficult to tell whether the approach is general enough for more specialists. As the real connection here is only through unification, it is

unlikely that other specialists could be added without major revisions.

LOGIN (Logic + Inheritance) [AiN86] is an enhancement to Prolog that allows type inheritance. The differences between this system and standard Prolog are in the language (additional constructs to describe the IS-A taxonomy) and the unification algorithm. Type inheritance was integrated directly into the unification process, separating it from the logical inference done by the system. The architecture of this system is quite similar to that of Krypton, and therefore generates the same question of how easy it would be to add different specialists.

The **Horne** system, developed by Allen et al. [AGF84], uses a special method which computes closure of a number of relational assertions, including temporal ones. While this allows constant time lookup later, this method restricts the assistance of the temporal specialist to the literal evaluation phase only. As their system requires that closure be done for each entry ($O(n^3)$ time) and has no facilities for retraction, generalized resolving and factoring in it would be inefficient, if at all possible.

The **ECoNet** system, which is the concern of this thesis, also falls into the present category of theorem-prover-based hybrids. It will be discussed in Chapter 4.

2.2. Blackboard Approaches

Blackboard approaches [EHL80, HWL83], like hybrids, combine several different representations and/or inference methods. These systems use a central shared data structure, called a blackboard, through which the specialists, or knowledge sources as they are generally referred to in this type of system, communicate. The blackboard is usually divided up into regions, most often corresponding to levels of abstraction, or hierarchically. Some examples of blackboard systems which divide their blackboards in this manner are HEARSAY, HASP, CRYSALIS (with

two hierarchies), and OPM (from Nii's comparison in [Nii86]). The knowledge sources cooperate and combine their efforts to solve the problem. They are opportunistic, activated when something of interest to them is written on the blackboard, and writing their conclusions back there for other sources. A scheduler coordinates the efforts of the specialists and tries to decide which of the activated ones is best to apply first. Many such systems (HEARSAY, HASP, OPM, and to a lesser extent TRICERO) use several problem solving approaches, including top-down, bottom-up, and generate-and-test strategies. HEARSAY especially must deal with noisy data and vague knowledge, and even contradictory knowledge sources, which adds complexity to both the sources themselves and the interactions. The solution is built incrementally, using many sources to confirm each step.

The major difference between blackboard systems and the hybrids in the previous section is that the hybrids have a formally interpretable representation - a logic. This leads to a different conception of specialist. Blackboard system knowledge sources usually consist of a set of ad hoc rules, and user defined operations, none of which are formally interpretable. Although this approach allows much flexibility in designing a system, it is really more of a programming methodology than an architecture per se. The problem we attempt to solve could have also have been tackled from a blackboard perspective, although the extra flexibility is not really required. The specialists themselves have strong theoretical backgrounds and do not conflict, and the operations are well defined logical operations. There is little trouble deciding which specialist(s) are applicable for a given literal or pair of them. Also, the hybrid approach restricts the operations a specialist is allowed to perform to the logical operations done by theorem provers (literal evaluation, determining incompatibility of two literals, etc), so we can ensure that the specialist is making logically sound inferences. By having a formally intepretable representation, we can also prove such a system complete, where in general, a blackboard system will not have this theoretical support.

Chapter 3

The Temporal Specialist

The temporal specialist, as the name suggests, does reasoning about time. Given certain explicit relationships among a set of events or episodes, we would like to be able to infer additional relationships implicit in the ones given. For example, if the events are part of a narrative, they will frequently form sequences in which adjacent events are known to follow one another; for such a sequence, we would like to be able to infer, without much effort, that events earlier in the sequence precede later ones, regardless of the number of intervening events. In addition, if we are given information about the durations or absolute times of some of the events, we would like to be able to infer quantitative consequences of this information, such as minimum and maximum elapsed time between given events. These sorts of inferences are essential in several areas of AI including story understanding, causal reasoning, and planning [All84], as well as in other areas, like scheduling, medical records, banks, etc.

We would also like to be able to represent and make inferences about absolute times (dates on a time line), which may be incompletely specified, and durations between events. As the system used in this thesis is intended for use in natural language understanding, and people generally speak of relative orders of events rather than exact times, the absolute time inference is of somewhat lesser importance than the relative ordering inference. A banking system, on the other hand, would consider reasoning with dates to be of the utmost importance. We are not concerned with such systems here however, as most Artificial Intelligence applications requiring temporal inference will have requirements similar to natural language understanding.

Our intuitive notions of time tell us that any theory of time should respect the transitivity axiom schema

$$\bigwedge_{i=1}^{n} [t_{i-1} \leq t_i] \Rightarrow [t_0 \leq t_n].$$

Furthermore, if any of the relations in the antecedent is strict, i.e. $[t_{i-1} < t_i]$, then the consequent is $[t_0 < t_n]$ (the relation between the first and last points is also strict). Some other basic properties of the time ordering are:

$[t_1 \leq t_2] <=> [t_1 < t_2] \mid [t_1 = t_2]$
$[t_1 < t_2] => \neg [t_2 \leq t_1]$
$[[t_1 \leq t_2] \ \& [t_2 \leq t_1]] => [t_1 = t_2]$ (a consequence of the previous properties)

Additional axioms defining the meaning of the predicates used by the temporal specialist in terms of $<$, $\leq$, and $=$ are in Section 3.2.2.

In fact, for the purpose of reasoning about the quantifiable aspects of time (dates and durations), it is natural to identify time with numbers, so that we can incorporate relevant properties of real numbers into our time theory. Thus, the relationship between durations and times satisfies:

$duration(t_1,t_2) = t_2 - t_1$
$t_1 \leq t_2 \leq t_3 => duration(t_1,t_3) = duration(t_1,t_2) + duration(t_2,t_3)$

We also have equations relating bounds on times and durations (from [Tau83], page 71). Given two consecutive time points t_1 and t_2 with lower and upper time bounds l_1, u_1, l_2 and u_2, respectively, and lower and upper bounds l and u on the duration between t_1 and t_2 $(t_2 - t_1)$, the following hold:

$l_1 \leq t_1 \leq u_1$
$l_2 \leq t_2 \leq u_2$
$l \leq t_2 - t_1 \leq u$
$u_1 \leq u_2$
$l_1 \leq l_2$
$t_1 \leq t_2$

For updating the bounds, the following consequences are used:

$l_2 - u \leq t_1 \leq u_2 - l$
$l + l_1 \leq t_2 \leq u + u_1$
$l_2 - u_1 \leq t_2 - t_1 \leq u_2 - l_1$

Clearly the above will all be satisfied if one interprets time as the real line. Proof that propagation of absolute time bounds establishes the best possible bounds throughout the time graph can be found in an appendix of Taugher's thesis.

Since temporal orderings are transitive and durations cumulative, such inferences in a typical general theorem prover can be computationally expensive. To compensate for this, researchers have tried to develop special representations and efficient methods for temporal inference. A summary of some methods of temporal representation and inference can be found in Taugher [Tau83]. The following focuses on research done subsequent to Taugher's work.

Most current research in temporal inference uses two basic types of representations - *time intervals* and *time points*. As Taugher noted, methods that directly represent absolute times are either not efficient or are incomplete for relative ordering, which has already been decided as being of primary importance here.

Time intervals represent a given time period of finite length. These can be thought of as directly representing the time in which an event or episode takes place. There are numerous simple relations (from [All83, ViK86]) that can be defined between two intervals (*equal*, *before* (including *meets*), *after* (including *met-by*), *during* (including *starts* and *ends*), *contains* (including *started-by* and *ended-by*), *overlaps*, and *overlapped-by*). More complex relations between intervals are disjunctions of these simple relations.

The other major representation used is *time points*. These are abstract "instants" of time and are assumed to be non-decomposable. Practically, there really is no such thing as a non-decomposable instant (even an instantaneous event can be divided into various parts), but it is a useful assumption to make for temporal inference. Pairs of such time points can be used to represent intervals. There are only a few relations that can be defined between time points ($<, \leq, >, \geq, =$).

Most other time representations are special cases of the above. For example, *reference intervals* [All83] are a special case of intervals that are broken down hierarchically and contain more intervals. *Non-convex intervals* [Lad86] are discontinuous, meaning the interval doesn't really include all the "points" within - there are gaps. Leban et al. [LMF86] use collections of

intervals.

There is a trade-off between expressiveness and efficiency in the use of the two representations. For instance, if we restrict the relationships that may be specificed for time intervals to *disjunctions* of those listed above (as Allen did), and similarly restrict relationships that may be specified for two time points to disjunctions of equalities and inequalities, then the *interval* representation is more expressive and flexible, while the *time point* one is more efficient. In particular, closure on Allen's interval based approach [All83] is shown to be NP-hard by Kautz and Vilain [ViK86], while closure may be done in $O(n^3)$ time in the time point representation, where n is the number of points. However, Schubert et al. [SPT87] suggest that the intractability problem results from allowing disjunctions of relations, rather than from using an *interval-based* approach. [Vil82] has a similar interval based system that does transitive closure in $O(n^3)$ time, but must backtrace through partial computations saved during computation of the transitive closure to find inconsistencies. If arbitrary conjunctive/disjunctive constraints on sets of time points are allowed, then of course the time point representation is as expressive as the interval representation, but efficiency is then lost. In general, the function of a specialist should be to handle as large a class of relationships as possible without sacrificing efficiency. Any inferences that fall outside the capabilities of the specialist can still be handled by the general deductive mechanism, albeit much less efficiently.

Taugher and Schubert [SPT83, SPT87, Tau83] developed a representation based on time points. This specialist is unusual in the way it exploits chains of events such as are commonly found in narratives (or plans), so as to achieve constant-time determination of time order for many pairs of events. It does so without requiring all events to lie on chains, and without computing transitive closure. Schubert et al. suggest that this *time point* representation can be used to determine inconsistencies in better than $O(n^3)$ time. An example of an inconsistency is *a before b, a before-1 c, c before a*, where *a* has been asserted as strictly before *c*, and also after *c*. Closure can

also be done in better time, although the worst-case asymptotic remains the same (see Appendix A2).

As an aside, closure is not a particularly efficient method to use for temporal reasoning unless all temporal relations are known before any questions are asked. If entry of new relations may occur at any point, the entire closure algorithm must be run to completion for each one, whereas Taugher's method, for example, usually does entry in constant time. There is a tradeoff here between efficient insertion and efficient lookup (once closure is done, all lookups are done in constant time), but the closure method is more restrictive in what environment it can efficiently operate in.

Since the *time point* representation is much more manageable than the *interval* representation, and furthermore, the Taugher and Schubert representation and inference method is more efficient than other *time point* representations (at least in narrative domains), the temporal inference specialist used in this thesis is based on that model.

To further increase the effectiveness of the specialist, its capabilities are extended here to make it more complete and flexible; the major enhancement being the ability to handle both strict and non-strict time ordering. Non-strict ordering is often appropriate for the end of one event in relation to the beginning of the next in a narrative (where there may or may not be a delay between them), while strict ordering is often appropriate for the beginning of an event in relation to its end (when the event is of a type that cannot transpire instantaneously). This, as well as consistency and expressiveness considerations, required considerable expansion of the set of temporal predicates and argument patterns handled by the specialist. Also, some additional reasoning is required of the specialist for generalized resolution and factoring.

3.1. The Taugher and Schubert Model

The temporal specialist implemented here, supporting the thesis that specialists are useful, is based on the specialist designed by Taugher and Schubert. Besides being able to determine relations between time points quickly (often in constant time), it can represent and reason with durations and absolute times. The representation uses a partial order graph, called a timegraph, whose nodes represent time points. Directed links between points indicate the given relation between the two points ($<$ or $\leq$ depending on interpretation). The timegraph is partitioned into *chains*, which are defined as sets of points that are all linearly ordered with respect to each other, with possible transitive arcs[3]. Links between points in the same chain are *in-chain* links; between points in different chains, *cross-chain* links. Each point has a *pseudo-time* (a number) associated with it, which is arbitrary except that it respects the ordering relationship between it and other points on the same chain. These numbers are generated by the system when a point is added. Determining the relationship between any two points in the same chain may be done in constant time simply by comparing their pseudo-times, rather than following the in-chain links.

To determine the relationship between points in different chains, a search is required, but only the cross-chain links need to be examined explicitly. A *metagraph* keeps track of the cross-chain links effectively by maintaining a *metanode* for each chain, and using the cross-chain links for links between metanodes. When searching for a path between points, the cross-chain links in the metagraph are used to go from one chain to another in the path, and within the chain, the pseudo-times are used to determine the relation. As in-chain checking can be done in constant time, a graph search is dependent on the number of cross-chain links rather than the total number of time points. (See Appendix A1 for an analysis of this difference.) The specialist makes efficient use of space as well, requiring $O(r)$ space, where r is the number of relations ($\ll n^2$,

[3] Schubert et al. clarify this point in [SPT87], as Taugher's thesis was not clear on it. Transitive arcs are in-chain links which do not make up the "backbone" of the chain; that is, there are intervening nodes and links. Figure 2.1 shows a transitive arc between circle2 and circle4.

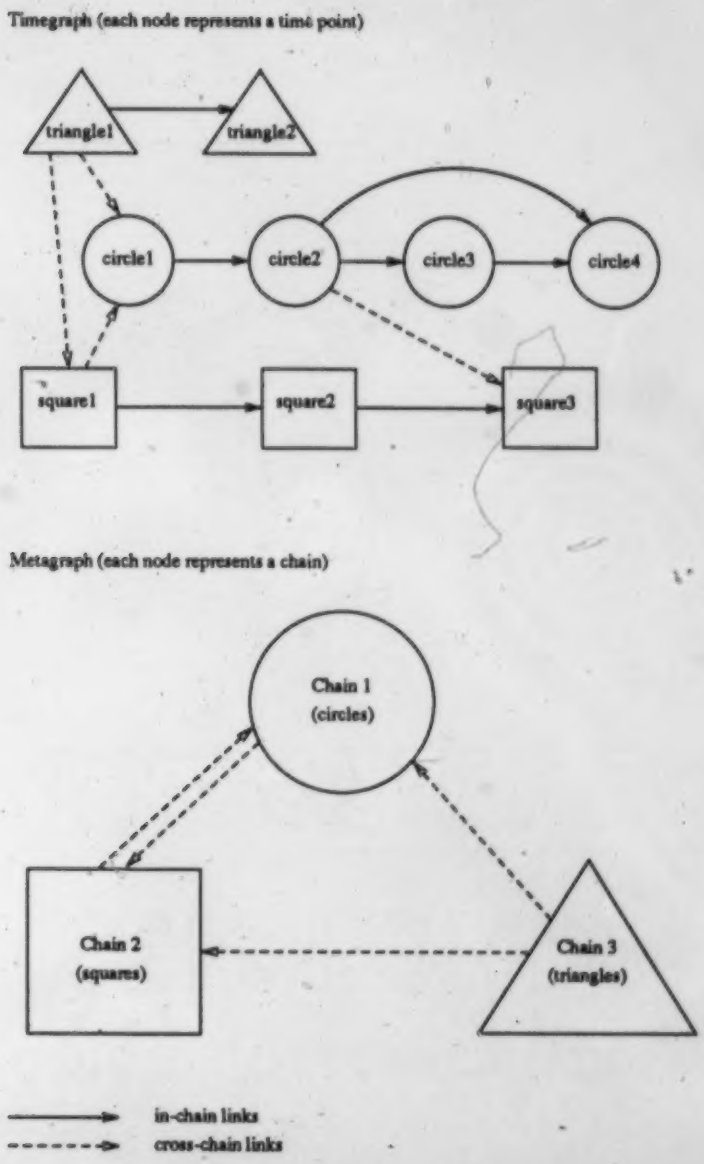

Figure 3.1 Example Timegraph and Metagraph

where n is the number of time points). Note that a timegraph amounts to a set of atomic inequalities. Disjunctions such as $[[a \leq b] \mid [c \leq d]]$ or inequalities such as $[a \neg= b]$ cannot be represented. This restriction, along with the exploitation of time chains, is what enables the

specialist to operate efficiently.

Figure 2.1 shows an example timegraph and metagraph. In the timegraph, small circles represent points on chain 1, small squares are points on chain 2, and small triangles represent points on chain 3. So circle1 is before circle2, circle2 before circle3 and so on. There are cross-chain links from triangle1 to circle1 (i.e. triangle1 is before circle1), from triangle1 to square1, from square2 to circle1, and one more from circle2 to square3. Following the cross-chain links, we can infer that triangle1 is before square3, and square1 is before circle3, but no information about the relation between triangle2 and square3.

A new chain is started whenever a point is added to the timegraph whose relation to other chains is not completely known. In the example time graph, if we now assert *[newpoint after square2]*, we must start a new chain for it, as the new points relation to every point on chain 2 is not known, and so we cannot maintain the linear relationship of points in a chain.

Furthermore, an absolute time (date) minimum and maximum are stored with each time point. These are provided by user assertions. They are a six-tuples of the form *(year month day hour minute second)*, where each element may be numeric or symbolic (e.g. *(1987 04 a 12 b c)* represents some time at or after 12 a.m. and before 1 p.m. of some day in April, 1987). Symbolic information may be filled in later by another assertion, or left unspecified throughout the session. Absolute time maxima propagate back to points before the given point (in the chain or on other chains), and minima propagate forward. This ensures that each point has the best absolute time information possible. Details may be found in [Tau83]. Absolute time comparisons can sometimes be used to get a relation in constant time between two points on different chains, avoiding a metagraph search.

Duration minima and maxima (in seconds) are stored on the links between points. These may affect the absolute times around them, which are then propagated. They are also used in calculating duration between points where the path uses this link.

Insertion into the graph requires a constant amount of processing in most cases, except for propagation of absolute times. In the worst case, propagation may require going to every point in the graph, although this is highly unlikely. Occasionally a chain may have to be renumbered, which requires going to all the points in a single chain.

In the metagraph, these cross-chain links show up as links between metanodes. There is one metanode for chain 1 (the big circle), one for chain 2 (the big square) and another for chain 3 (the big triangle). The links within chains do not show up here, as within a chain they are not needed to determine relations.

Intervals/events/episodes[4] are represented by their start and end time points. The intervals are kept in a separate table, which contains these time points. Interval relations are defined as boolean combinations of time point relations between the start and end points of the intervals.

3.2. Changes to the Original Temporal Specialist

Although the original Taugher time model is efficient and does most of the temporal reasoning we need, some improvements were necessary to increase its expressiveness and to enable it to do more in a resolution environment. First, confusion surrounding whether a point (*a*) could be inserted between two others (*b* and *c*) and remain on the same chain[5], was resolved by allowing this to take place only when *c* has the smallest pseudo-time of any in-chain descendant of *b* (i.e is closest). Note that in this case the link from *b* to *c* will be a transitive arc (bypassing one intermediate node, namely *a*) of the resulting chain. Otherwise *a* must be placed on a new chain.

Second, the original representation can represent either strict relations (<, >, =), or non-

---

[4] Interval, event, and episode are used interchangeably in this thesis, although they are quite different entities. Lacking a comprehensive theory of episodes and events, we simply treat them as the time intervals they occur in.

[5] A CMPUT 551 (Artificial Intelligence I) class at the University of Alberta discovered this in 1984 while reimplementing the algorithm as part of a project.

strict relations (≤ , ≥, =), depending on whether links between points are interpreted as < or ≤ , but not both. Our goal here was to extend the model to handle both, without losing the efficiency of the original. This, as well as consistency and expressiveness considerations necessitated changes to some of the argument patterns used.

3.2.1. Strict and Non-Strict Relations

The original Taugher representation can represent either all strict relations (<, >, =), or non-strict relations (≤ , ≥, =), depending on whether links between points are interpreted as < or ≤ , but not both. Non-strict ordering is often appropriate for the end of one event in relation to the beginning of the next in a narrative (where there may or may not be a delay between them), while strict ordering is often appropriate for the beginning of an event in relation to its end (when the event is of a type that cannot transpire instantaneously). Non-strict relations can be viewed as a limited form of disjunction - for example, *before* represents ≤ and so the relation may be strict (<) or equal. As both of these orderings are used in narrative understanding, the time specialist needs to be extended to represent and reason with both strict and non-strict orderings. Also, given additional information, a previously non-strict relation may be actually be strict, and we need some way to ensure that this additional constraint is detected and used. We always maintain the most constrained, but correct information possible in the time graph. We later may constrain it even more, but we cannot relax any of the constraints represented in the timegraph.

To represent the strict/non-strict difference, there are two types of relations in the timegraph to consider. For cross-chain relations, a flag indicating strictness on the link itself is enough, since cross-chain links are explicitly examined during a search. We can take note of the flag as we search for a path through the cross-chain links - a strict link anywhere on the path means the whole path represents a strict relation, otherwise the relation is non-strict.

For points within the same chain however, examining each link for strictness would mean

that we could no longer determine in-chain relationships in constant time. One possibility is to store another "pseudo-time" with each point, called an *equal-pseudo*. If any two points have the same *equal-pseudo* times, the relation given by their pseudo-times is non-strict, otherwise it is strict. Whenever a strict relation is asserted (either for the first time, or re-asserting a relations which was previously non-strict), the *equal-pseudos* of the two points involved are changed to their pseudo-times to make them unique, and the new *equal-pseudos* propagated if necessary. Propagation is done by replacing *equal-pseudos* equal to the old value with the new one, going forward from the latter point, and backward from the first point. For example, if $a \leq b \leq c \leq d$:

Point	*a*	*b*	*c*	*d*
Pseudo-time	1	1000	2000	3000
Equal-pseudo	1	1	1	1

After asserting b < c:

Point	*a*	*b*	*c*	*d*
Pseudo-time	1	1000	2000	4000
Equal-pseudo	1000	1000	3000	3000

There are some problems with this approach though - we cannot capture all possible combinations of relations. For example, if $a \leq b \leq c$ all *equal-pseudos* are the same. If we then assert $a < c$, b should still be $\leq c$, and $\geq a$, and therefore its *equal-pseudo* must be equal to c's and also to a's. However, that means a and c must have the same *equal-pseudo*, which contradicts $a < c$.

It may be possible to allow the *equal-pseudo* to take on two values in such cases - a's *equal-pseudo* and b's *equal-pseudo*. A point before it on the chain would have its *equal-pseudo* compared against the first *equal-pseudo*, and point after it against the last (the order on the chain is known via the pseudo-times). However, it is easy to construct a scenario where this will not be good enough. For example, $a \leq b \leq c \leq d \leq e$ (all have same *equal-pseudos*). Now assert $a < d$, giving

Point	*a*	*b*	*c*	*d*	*e*
Pseudo-time	1	1000	2000	3000	4000
Equal-pseudo	1	1,3000	1,3000	3000	3000

This has $a < d$, $a \leq b$, $b \leq c$, $a < e$, and so on - correctly. But what happens if we also assert $c < e$?

Point	*a*	*b*	*c*	*d*	*e*
Pseudo-time	1	1000	2000	3000	4000
Equal-pseudo	1	1,2000	2000	2000,4000	4000

This has maintained $a < d$, and represented $c < e$, but now $a < c$ (and it should be $\leq$). It is also difficult to decide how to change and propagate the *equal-pseudos*.

The successful method eventually found involves two extra numbers similar to the *equal-pseudos* - a *maximum-pseudo* and a *minimum-pseudo*. The *minimum-pseudo* of a given point is the pseudo-time of the nearest predecessor on the chain that the point cannot be equal to; the *maximum-pseudo*, of the nearest successor on the chain that it cannot be equal to. This requires the use of two additional pseudo-times to represents points before the beginning of the chain ($-\infty$) and beyond the end ($+\infty$). To make chain renumbering easier, pointers to the minimum and maximum points are kept, rather than their actual pseudo times.

Now to determine the relation between two points, we first compare their pseudo-times to find out what order the points are in. Then we compare the pseudo-time of the first point with the range given by the minimum and maximum of the second point (or vice versa). If it is properly within that range (i.e. greater than the minimum and less than the maximum), the relation given by the pseudo-times alone is non-strict; otherwise it is strict. Propagation of minima and maxima maintains strictness of relations throughout the chain. This propagation is similar to propagation of absolute times (minima propagate forward, maxima back), although the comparison at each propagation step may be shorter (only one number to compare instead of a possible six).

Using this representation on the previous problem, we start with $a \leq b \leq c \leq d \leq e$, and assert $a < d$:

Point	*a*	*b*	*c*	*d*	*e*
Pseudo-time	1	1000	2000	3000	4000
Minimum-pseudo	$-\infty$	$-\infty$	$-\infty$	1	1
Maximum-pseudo	3000	$+\infty$	$+\infty$	$+\infty$	$+\infty$

Again, we have $a < d$, $a \leq b$, $b \leq c$, $a < e$, and so on - correctly. Now assert $c < e$:

Point	*a*	*b*	*c*	*d*	*e*
Pseudo-time	1	1000	2000	3000	4000
Minimum-pseudo	$-\infty$	$-\infty$	$-\infty$	1	2000
Maximum-pseudo	3000	4000	4000	$+\infty$	$+\infty$

Note that $a \leq c$ now, as *c*'s *pseudo-time* of 2000 is within the interval given by *a*'s *minimum-pseudo* of $-\infty$ and *a*'s *maximum-pseudo* of 4000. The other relations are maintained as well.

This method is flexible enough to handle any combination of relations, easy to understand and implement, and maintains the constant time in-chain checking.

To evaluate whether a given relation holds, or to determine the strongest relation constraining two points, we now need to consider that there may be numerous paths between the points, some of which represent strict relations (a strict link anywhere along the path), and some non-strict (no strict links). To determine the relation between two points, one must continue finding paths until a strict path is found or all such paths have been found. Within each chain, the strictness of the relation between points is easily calculated in constant time (the range comparison), and used in the path determination. It is only where there are several possible cross-chain links leaving a point that there is a possibility for paths of different strictness. In the worst case, this could require $O(m^2)$ time, where m is the number of cross-chain links. Since we stop with the first strict path found, this maximum will rarely be reached. With some additional effort, we could improve the algorithm to save partial results between points so that each cross-chain link need only be looked at once. This would bring the time required back to $O(m)$, the same as for a search without strictness requirements.

3.2.2. Predicates and Argument Patterns

With the additional capabilities for handling strict and nonstrict ordering, we now need some way of telling the temporal specialist which ordering to use. This was one reason, among others, for some changes to the set of temporal predicates recognized by the specialist. In Taugher's thesis, the following predicates were used for assertion (entry) of intervals/events:

a before b [c]	=>	*a end* ≤ *b start* (both within time frame *c* if specified)
a after b [c]	=>	*a start* ≥ *b end* (both within time frame *c* if specified)
a equal b	=>	*a end* = *b start*
a during b [c]	=>	if *c* specified, *a between b and c* otherwise *a within b*

On evaluation, some additional relations could be determined: *overlaps*, *overlapped-by*, *contains*, as well as some weaker ones: *starts-before*, *ends-before*, *starts-after*, *ends-after*, *starts-equal*, and *ends-equal*.

The predicates used in the temporal specialist implemented here are similar to those of Taugher, with some exceptions. The weaker predicates have not been implemented. *During* is used only in the sense of "within", enabling the option of a timeframe argument. *Between* was introduced to handle the case where one argument is to be inserted "during" two others (see above). Except for *between*, all predicates now have the optional third argument representing a time frame (instead of just *before* and *after*). Usage of this argument leads to fewer chains being built in the timegraph, resulting in a more nearly optimal timegraph. This is because the timeframe argument *c* usually exists, while the other arguments don't necessaarily. By relating the new temporal entities to existing ones, we can reduce the number of chains which have to be built. All the predicates may be used for both entry (assertion) and evaluation. This provides a more uniform and consistent interface for a user.

To allow the user to specify whether to use strict or non-strict ordering, the original predicates are now considered "stems" and may have one or two strictness values appended (*stem[-strict1[-strict2]]*). Strictness values are:

Strictness Value	Meaning
-1	strict (< or >)
-0	meets (end points abut - i.e. are equal)
nil	non-strict (≤ or ≥).

Equal requires no strictness values, as the predicate implies equality of end points. *Before* and *after* involve only one relation (between the end of one and the start of the other argument), and so have one strictness value associated with them. For example, *before* means ≤ , *before-0* is *meets*, and *before-1* is <.

Between involves two relations, and so has two strictness values. For example, *[a between b c]* means *[a after b]* and *[a before c]*, so the first strictness value is for the *a-b* relation, and the second for the *a-c* relation. For example, *[a between-0-1 b c]* means *[a after-0 b]* & *[a before-1 c]* (*i.e. start of a = end of b*, and *end of a < start of c*).

The other predicates also involve two relations, between the start points of the two events, and the end points of the two events. They also have two strictness values. For example, *[a during--1 b]* means *start of a ≥ start of b*, and *end of a < end of b*.

In terms of the temporal ordering relations <, ≤ and =, the following axioms define the temporal predicates above (arguments beginning with *e* represent intervals (events); with *t*, time points):

$$
\begin{array}{lcl}
[e_1\ PRED\ e_2 e_3] & <=> & [e_1\ PRED\ e_2]\ \& \\
\text{PRED is one of:} & & [(start\text{–}of\ e_3) \le (start\text{–}of\ e_1)]\ \& \\
before, after, during, & & [(end\text{–}of\ e_1) \le (end\text{–}of\ e_3)]\ \& \\
contains, overlaps, & & [(start\text{–}of\ e_3) \le (start\text{–}of\ e_2)]\ \& \\
overlapped\text{–}by & & ![(end\text{–}of\ e_2) \le [(end\text{–}of\ e_3)] \\
 & & \\
[e_1\ equal\ e_2] & <=> & [(start\text{–}of\ e_1) = (start\text{–}of\ e_2)]\ \& \\
 & & [(end\text{–}of\ e_1) = (end\text{–}of\ e_2)] \\
[t_1\ equal\ t_2] & <=> & [t_1 = t_2] \\
 & & \\
[e_1\ before\ e_2] & <=> & [(end\text{–}of\ e_1) \le (start\text{–}of\ e_2)] \\
[e_1\ before\text{–}1\ e_2] & <=> & [(end\text{–}of\ e_1) < (start\text{–}of\ e_2)] \\
[e_1\ before\text{–}0\ e_2] & <=> & [(end\text{–}of\ e_1) = (start\text{–}of\ e_2)] \\
[t_1\ before\ t_2] & <=> & [t_1 \le t_2] \\
[t_1\ before\text{–}1\ t_2] & <=> & [t_1 < t_2]
\end{array}
$$

$[t_1\ before\text{–}0\ t_2]$ <=> $[t_1 = t_2]$

$[e_1\ after\ e_2]$ <=> $[(end\text{–}of\ e_2) \leq (start\text{–}of\ e_1)]$
$[e_1\ after\text{–}1\ e_2]$ <=> $[(end\text{–}of\ e_2) < (start\text{–}of\ e_1)]$
$[e_1\ after\text{–}0\ e_2]$ <=> $[(end\text{–}of\ e_2) = (start\text{–}of\ e_1)]$
$[t_1\ after\ t_2]$ <=> $[t_2 \leq t_1]$
$[t_1\ after\text{–}1\ t_2]$ <=> $[t_2 < t_1]$
$[t_1\ after\text{–}0\ t_2]$ <=> $[t_2 = t_1]$

$[e_1\ during\ e_2]$ <=> $[(start\text{–}of\ e_2) \leq (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) \leq (end\text{–}of\ e_2)]$
$[e_1\ during\text{–}1\ e_2]$ <=> $[(start\text{–}of\ e_2) < (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) \leq (end\text{–}of\ e_2)]$
$[e_1\ during\text{–}0\ e_2]$ <=> $[(start\text{–}of\ e_2) = (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) \leq (end\text{–}of\ e_2)]$
$[e_1\ during\text{–}1\text{–}1\ e_2]$ <=> $[(start\text{–}of\ e_2) < (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) < (end\text{–}of\ e_2)]$
$[e_1\ during\text{–}0\text{–}1\ e_2]$ <=> $[(start\text{–}of\ e_2) = (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) < (end\text{–}of\ e_2)]$
$[e_1\ during\text{—}1\ e_2]$ <=> $[(start\text{–}of\ e_2) \leq (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) < (end\text{–}of\ e_2)]$
$[e_1\ during\text{–}1\text{–}0\ e_2]$ <=> $[(start\text{–}of\ e_2) < (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) = (end\text{–}of\ e_2)]$
$[e_1\ during\text{–}0\text{–}0\ e_2]$ <=> $[(start\text{–}of\ e_2) = (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) = (end\text{–}of\ e_2)]$
$[e_1\ during\text{—}0\ e_2]$ <=> $[(start\text{–}of\ e_2) \leq (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) = (end\text{–}of\ e_2)]$

$[e_1\ contains\ e_2]$ <=> $[(start\text{–}of\ e_1) \leq (start\text{–}of\ e_2)]$ & $[(end\text{–}of\ e_2) \leq (end\text{–}of\ e_1)]$

$[e_1\ overlaps\ e_2]$ <=> $[(start\text{–}of\ e_1) \leq (start\text{–}of\ e_2)]$ & $[(end\text{–}of\ e_1) \leq (end\text{–}of\ e_2)]$

$[e_1\ overlapped\text{–}by\ e_2]$ <=> $[(start\text{–}of\ e_2) \leq (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_2) \leq (end\text{–}of\ e_1)]$

$[e_1\ between\ e_2 e_3]$ <=> $[(end\text{–}of\ e_2) \leq (start\text{–}of\ e_1)]$ & $[(end\text{–}of\ e_1) \leq (start\text{–}of\ e_3)]$

Note that this assumes the existance of functions *start–of* , which returns the time point corresponding to the start of an interval, and *end–of* , which returns the time point corresponding to the end of an interval. For brevity, not all prediates are represented here: *contains*, *overlaps*, *overlapped-by*, and *between* have stem endings like *during*. Axioms for these predicates with strictness indicators can be determined by noting the effect of the strictness indicators on the

during axiom.

The new specialist is quite flexible, accepting intervals, time points, or absolute times in any combination as arguments for the predicates (except the timeframe, which must be an episode), where Taugher's programs were quite rigid in the argument patterns accepted. On assertion, if an argument is an absolute time instead of a named time point or episode, the appropriate absolute time bound of the other argument is updated. For example,

(a before (date '1987 '04 '01 '00 '00 '00))

would update the upper bound of *a*'s absolute time (the maximum), while

(a after (date '1986 '01 '01 '00 '00 '00)) would change its lower bound. For evaluation, the appropriate bound is compared against the absolute time, or two absolute times may be compared.

In addition, some new predicates were invented to handle durations: *at-most-before*, *at-most-after*, *at-least-before*, *at-least-after*, *exactly-before*, and *exactly-after*. These take three arguments, of which the first two may be events or time points (no absolute times) and the third denotes the duration between the two in seconds. *At-most-* implies maximum duration, *at-least-* minimum duration, and *exactly-* involves both. Evaluation of durations requires a full blown graph search to find all possible paths and the durations associated with them. This calculation will be described shortly.

3.2.3. Duration Determination

To determine the duration between any two points, an exhaustive search must be done to find all paths from one point to the other. This particular search follows all links, not just cross-chain links (i.e. it doesn't just use the metagraph). The duration is calculated along all paths to get the best duration bounds (the greatest minimum (lower bound), and the smallest maximum (upper bound)). Both duration information on arcs and duration information implicit in absolute times

are used. The algorithm is described in Figure 2.2.

```
determine-duration(pt1 pt2 curmin curmax already)
    bestmin = −∞
    bestmax = +∞
    For each link (in-chain and cross-chain) leaving pt1:
        If link "to point" is in already, ignore
        Calculate link duration by taking best of duration on links and implied
            duration using absolute times
        newmin = curmin + link duration minimum
        newmax = curmax + link duration maximum
        If link "to point" is pt2
            bestmin = (max bestmin newmin)
            bestmax = (min bestmax newmax)
        Else
            (determine-duration to-point pt2 newmin newmax
                (append already "to point"))
```

Figure 3.2 Algorithm to Determine Duration

3.3. Use of the Temporal Specialist

Further extensions were necessary to integrate the specialist into the main system, which is described in chapter 4. The main system organizes modal propositions into subnets, with one subnet for each person's mental world. The subnets may contain information that holds only in certain "possible worlds" (such as a fairy-tale, or someone's mental world), but not in the real world, so the temporal specialist also maintains a separate timegraph (and metagraph) for each subnet. This assumes that all temporal propositions within a mental world are consistent - i.e. that John's temporal beliefs are consistent with his hopes, and so on. Simple evaluations may be done within the subnets, although at this stage no modal inference is done by either the temporal specialist or the main theorem prover.

Upon completion of proofs by contradiction, the main system retracts all changes made to the logical representation (semantic net). To remain consistent, the temporal specialist had to be extended to have this capability as well, which is used extensively in generalized resolving and factoring. Changes are stored at an atomic level, and may only be retracted in reverse order.

The specialist handles literal evaluation, generalized resolving and factoring, and, in addition, can be used to simplify literals by evaluating functional terms. Assertions must also be entered into the specialist representation for use in future evaluations. Generalized resolution and factoring make use of both the entry and evaluation phases to check for resolving or factoring actions. Entry and evaluation of temporal literals is done only for literals with constant arguments, although during generalized resolution and factoring, entry and evaluation are extended to treat variables as constants. Although during the entry phase, only literals with constant arguments will be inserted into the time graph, during generalized resolution, this is extended so that one might get a time point for the variable x in the time graph. This is later retracted anyway, but allows literals with variable arguments to be resolved against. Only positive (i.e. non-negated) literals are entered, but negated ones may be evaluated and used in generalized resolving and factoring.

Inferences made by the specialist are essentially applications of axioms of time theory. For example, the ordering inferences are based on the transitivity axiom stated at the beginning of this chapter. As we shall see shortly, generalized resolution and factoring embody some other useful axioms. Because the inferences are basically resolution inferences with the temporal axioms, the specialist inherits the soundness of resolution.

Completeness could be also proven for certain classes of premises for this specialist - for example, it is clearly complete for $\leq$ and $<$ for arbitrary sets of ground inequalities. These ordering inferences are the main function of the temporal specialist. In outline, the proof is as follows. The algorithm confirms $t_0 \leq t_n$ for any two of the given time points t_0, t_n, just in case there is a path from t_0 to t_n. By the transitivity axiom, this inference is sound. Moreover, suppose that this procedure is not complete, so that the ground inequalities represented by the graph along with the time axioms, logically entail $t_0 \leq t_n$, but there is no path from t_0 to t_n. But it is an elementary property of acyclic directed graphs that for any two vertices not connected by a directed path, one

can insert an edge from either of these vertices to the other, without creating any cycles. Since any acyclic graph is consistent with the time axioms (e.g. there is always a real number interpretation of the vertices satisfying the ordering constraints, even when all of them are interpreted strictly), it follows that $t_n < t_0$ can be consistently added to the premises, and hence $t_0 \leq t_n$ cannot be a consequence.

Furthermore, the algorithm confirms $t_0 < t_n$ just in case there is a path from t_0 to t_n containing a strict ordering link. Again soundness follows from (the strict version of) the transitivity axiom. If there is no path from t_0 to t_n, completeness follows as above. If there are paths, but they consist entirely of $\leq$ links, then all these paths can be collapsed into a single vertex without creating any cycles in the time graph as a whole (again a property of acyclic graphs). Hence t_0 can be interpreted as equal to t_n without inconsistency (e.g., using real numbers as a model), and so $t_0 < t_n$ cannot be a consequence.

We will not attempt to establish any further completeness claims here, as this is not the emphasis of this thesis. Doing so would prevent us from exploring other aspects - the interface design, and multiple specialists. The objective is to **accelerate** inference in cases that frequently arise in practice (such as transitive inferencing across ground inequalities), and for this objective, the issue of completeness with respect to certain restricted classes of premises is somewhat irrelevant.

3.3.1. Function Evaluation

Some temporal functions have been defined that may be used in propositions. During literal simplification, these functional terms may be evaluated to a constant or quoted expression that is more useful than the original form.

Note: all arguments of the function are evaluated first (possibly recursively) and must be constants. For simple assertions, where the arguments to the function are constants, this might

seem trivial, but when the functional arguments are variables, a later resolution might find a constant binding for them, and the function can then be evaluated and used to further simplify the clause.

The functions are[6]:

> *(***start-of** *E1)* - returns the concept corresponding to the start point of an event (e.g. E1start). The resulting concept is of sort *time*.
>
> *(***end-of** *E1)* - returns the concept corresponding to the end point of an event (e.g. E1end). The resulting concept is of sort *time*.
>
> *(***date** *year month day hour minute second)* - returns a quoted expression of the form *'(time year month day hour minute second)* where the numeric value of each argument is used if known, and if not, the name of the placeholder is used (e.g. *'(time 1987 mm 01 12 00 00)* - noon on the first day of an unspecified month in 1987).
>
> *(***relation** *a b)* - returns the most strict relation existing between the two arguments *a* and *b*. The relation returned is a concept.
>
> *(***duration** *a b)* - returns the minimum and maximum duration between the arguments *a* and *b* in a quoted expression *'(minimum maximum)*.

3.3.2. Literal Evaluation

When evaluating a literal (for example, is *[e1 during e2]*?), one or more of the evaluation techniques already discussed is used. If the predicate involved is a duration predicate (*at-most-before*, etc), a graph search is done to find out the duration and compare it to the one given. This relects the cumulative nature of the quantifiable aspects of time. If this gives no conclusive result, the two temporal arguments are compared to see if they are in the other order (to see if the relative order of the arguments is reversed), in which case a false answer is returned. For other predicates, if any arguments are absolute times, an absolute time comparison is done, much like comparing two numbers on a real number line.

---

[6] The functions return concepts or quoted expressions, with associated sorts. These will all be defined in Chapter 4, with the main theorem prover.

Otherwise, the question deals with temporal ordering, and the decision process must take into account the transitive nature of the relations. To handle intervals, the question is split into several time point evaluation questions, each of which is done by first determining the existing relation between the points using an in-chain check or a metagraph search. This relation is then compared to the one in the literal being evaluated. This ensures that yes or true is returned if the relation in the graph is subsumed by the questioned relation (for example, asking if *(a before b)* when *(a before-1 b)* is represented in the timegraph). All point evaluations must be *true* for the corresponding interval question to be *true*; any *false* point evaluation makes the interval evaluation *false*, and any *unknown* point evaluation makes the interval evaluation *unknown*.

3.3.3. Generalized Resolution and Factoring

Schubert, et al. [SPT87] show some examples of when generalized resolving and factoring can be done by a temporal specialist. Although some of the inferences made for generalized resolution and factoring are not as "obvious" as simple evaluations are, they do reflect the properties of temporal relations mentioned earlier, such as transitivity. Some examples of generalized resolving (from [SPT87]):

$$[t \leq t'] \text{ against } [t' \leq t]$$

gives a residue[7] of $[t = t']$. This inference is equivalent to two ordinary resolution steps based on the antisymmetry axiom

$$[t_1 \leq t_2] \,\&\, [t_2 \leq t_1]] \Rightarrow [t_1 = t_2]$$

Another example, using additional information from the timegraph:

$$\neg [c \leq t] \text{ against } [c' \leq t]$$

along with timegraph information $[c \leq c']$, gives the null clause. This can be determined by resolution inferences with the transitivity axiom, $[c \leq c']$ and $[c' \leq t]$, giving $[c \leq t]$, which directly

[7] A residue (from Stickel's partial theory resolution [Sti83] is a literal (or set of literals) whose negation would make the two literals incompatible (resolvable in one or more steps to the null clause).

contradicts $\neg[c \leq t]$, giving the null clause. An example of generalized factoring, using timegraph information:

$$[c \leq t] \mid [c' \leq t]$$

again with timegraph knowledge that $[c \leq c']$, gives generalized factor $[c \leq t]$, which can be obtained in two ordinary resolution steps and one ordinary factoring step by applying the transitivity axiom. (However, note that the preceeding examples of generalized resolution and factoring may in a sense condense arbitrarily many ordinary resolution steps, since the relationship between c and c' may be a consequence of arbitrarily many facts stored in the timegraph.)

There are many more cases where such operations can be done, dependent mainly on information already asserted in the timegraph. This section shows a method general enough to capture all useful cases of generalized resolving and factoring. The temporal specialist determines possible unifications that may lead to resolving or factoring actions, and uses the timegraph as a medium in which to compare the literals after substitution.

Determining whether two literals are resolvable or factorable is done using similar methods, so they will be described together. When resolving or factoring, unification is required. Since the predicates in the two literals are not identical (otherwise the temporal specialist would never have been called), there is no restriction that the arguments be unified in the same order (i.e. first from literal1 with first from literal2, and so on). Sometimes that may give no information at all. There are other ways that the arguments can be unified that may result in a resolving or factoring action. All possible unifications are tried here, with evaluations used after each to determine whether they are worthwhile. In determining unifications, event/time point arguments may only be unified with the same (i.e. not against durations), and the two literals do NOT need to have the same number of arguments[8].

[8] The only restriction on the unifications is that the timeframe argument, if present, not be unified with anything. That argument is used on entry only to generate a more nearly optimal timegraph, and is not essential for later reasoning - the relation given by the first two arguments with the predicate is the important one. This means fewer unifications to be tried for factoring or resolution and therefore these attempts will be fas-

Examples:

Literals	Unifications
(e1 before e2 e3) , (x after-1 y)	(e1/x, e2/y), (e2/x, e1/y)
(x after y) , (e1 between e2 e3)	(e1/x, e2/y), (e1/x, e3/y), (e2/x, e1/y), (e2/x, e3/y), (e3/x, e1/y), (e3/x, e2/y)

Each unification is like a shortcut for applying a number of temporal axioms to the literals to make them resolvable or factorable in the traditional sense. For instance, in the first example above, the unification (e2/x, e1/y) could be obtained by applying the temporal axioms given earlier to define *before* and *after*−1, as well as the ordering axioms about < and ≤ , to get literals $[(end\text{-}of\, e_1) \leq (start\text{-}of\, e_2)]$ and $[(end\text{-}of\, y) \leq (start\text{-}of\, x)]$, which can be unified in the traditional manner, with unifier (e2/x, e1/y). (This is the second of the two unifiers listed above. The first is obtained similarly, from inferred literals $[(end\text{-}of\, e_1) \leq (start\text{-}of\, e_2)]$ and $\neg[(end\text{-}of\, x) \leq (start\text{-}of\, y)]$.)

During a generalized resolving attempt, we are looking for a unification that makes the two literals incompatible, or incompatible with the negation of a residue. If the two relations are disjoint, we have the usual case of resolution with a null residue. If another relation holds however (subsumes, intersects), we don't have as strong a result, but we can still get something - with a residue. During generalized factoring, we look to see if there is a unification in which we may throw out one literal without losing anything (i.e. one is subsumed by the other, or one evaluates to false (or the other to true)).

For each unification, we try to enter one literal into the timegraph, and compare the other one to it within the timegraph. Use of the timegraph allows comparison against the newly entered

---

ter.

literal (what we started out to do), as well as other temporal relations asserted earlier (which enables the time specialist to shortcut the proof so drastically).

For each possible unification, the following steps are taken:

1. The substitutions ($s_1,...,s_n$) are made in the two literals

   This must be done first so that we are working with the special case of the literals, using the unification. These are the literals that the temporal specialist will try to resolve or factor, not the original literals.

2. Both literals are evaluated

   It may be possible to determine just by evaluation whether the literals are resolvable or factorable. In addition, if one evaluates to false, we certainly don't want to enter it into the timegraph and make it inconsistent.

```
For resolution:
  If not (value literal1) = unknown
    Resolvable with evaluations[9] (value literal1) and (value literal2)
      and substitutions (s1,...,sn)
    If both literals evaluated
      finished this resolving attempt - stop
    Else
      continue looking for other resolving actions involving residues

For factoring:
  If (value literal1) = true or
    (value literal2) = false
    Factorable with literal1 as the factor, and substitutions (s1,...,sn)
        - stop
  Else If (value literal1) = false or
       (value literal2) = true
       Factorable with literal2 as the factor, and substitutions (s1,...,sn)
          - stop
```

[9] This resolving action will be described in Chapter 4.

If this gives nothing, we continue:

We now try to enter one literal and compare it against the other within the timegraph. Other than negated information, which cannot be entered into the timegraph, it does not really matter which literal is entered and which tested - they should show the same effects either way. However, for resolving, when the residue is calculated, we would prefer it to have terms from the set of support clauses, and since the residue is determined based on the testing literal, we prefer that literal1 (which is always from the set of support or resolvents) be tested, and literal2 entered. There are exceptions to this, described below. For factoring, we attempt to enter the first one and test the second, and then the second one and test the first, so both combinations are tried.

3. The timegraph is checkpointed

This ensures that we can retract any information put in, as the addition of information by the proof attempt and the disproof attempt may be incompatible and make the timegraph inconsistent.

4. One literal is temporarily entered into the timegraph

The next step is to enter one literal into the timegraph so that the other can be tested against it. This ensures that we can detect inconsistencies between the two literals alone, even if we have an empty timegraph. Given a choice of literals to enter (i.e. none has been ruled out because of considerations described previously), if one has a *between* predicate, it is entered. This is because testing a *between* predicate is compound and therefore more complicated than any of the others.

5. The other literal is tested **(a1 pred1 a2 a3)**

There are two ways to test the other literal - a simple evaluation, or a comparison of the literal relation and the existing relation in the time graph. The former possibility leads to

either a null resolvent, or no information for resolving, while the latter can give resolvability with a residue. The latter is used whenever possible because it gives more information. There are some cases however, in which it cannot be used (for example, with duration predicates). A residue is determined as a literal whose negation would make the two literals we are resolving incompatible. This is done by comparing the predicate we are testing **pred1** with the relationship existing between the same arguments (**a1, a2**) in the timegraph. If there is an overlap between the two relations, the residue is that overlap; if no overlap, there is no residue. For example, if the two relations are disjoint, there is no overlap, and we get resolvability with a null residue. If the testing relation was *[a before b]*, and we have *[a after b]* in the timegraph, the overlap is *equal*, so the residue will be *[a equal b]*. A testing relation of *[a before b]*, and timegraph relation of *[a before-1 b]* would result in a residue of *[a before-1 b]*[10]. This is trickier to calculate when one literal is negated - in that case, one must work with the difference between the two relations, instead of their overlap. For example, testing relation $\neg$*[a equal b]* with timegraph relation *[a before b]* would result in a residue of *[a before-1 b]*.

If factoring
  If (value test-literal) = true
    Factorable with test-literal as the factor,
      and substitutions $(s_1, \ldots, s_n)$
  Else If (value test-literal) = false
      Factorable with the entered literal as the factor,
        and substitutions $(s_1, \ldots, s_n)$

If resolving
  For testing literals with a duration predicate (*at-most-*, etc):
    If (value test-literal) = false
      Resolvable with a residue of nil,
        and substitutions $(s_1, \ldots, s_n)$
    Else Not Resolvable - stop

  For testing literals with a *between* predicate:

[10] When a residue is calculated, it must be compared against the original literals to make sure it is not the same - if so, no resolution action is added.

Split the test into two tests **(a1 after+ a2)** and **(a1 before+ a3)**,
where **after+** and **before+** are predicates determined by the stems on the testing predicate (for example, *between-0-1* would split into tests *after-0* and *before-1*).
Do the next case (the otherwise case) for both of the tests, and combine the results.

Otherwise the predicate in **(a1 pred1 a2)**, and the existing relationship in the timegraph **(a1 pred2 a2)** are compared.
For a non-negated literal:
Case comparison between testing and graph relations in[11]:

disjoint	Resolvable with a residue of nil
subsumes	Resolvable with a residue of **(a1 pred2 a2)** (the graph relation)
subsumed	Resolvable with a residue of **(a1 pred1 a2)** (the testing relation)
equivalent	Not Resolvable
intersects	Resolvable with a residue of **(a1 equal a2)**

For a negated literal:
Case comparison between testing and graph relations in:

disjoint	Not resolvable
subsumes	Resolvable with a residue of nil
subsumed	Resolvable with a residue of **(a1 pred3 a2)**, where **pred3** is the difference between the two relations (e.g. difference between before and before-1 is equal).
equivalent	Resolvable with a residue of nil
intersects	If **pred2** is *equal*, Resolvable with a residue of nil Else Resolvable with a residue of **(a1 pred3 a2)** where **pred3** is the more strict version of **pred2**

6. Any changes made above are retracted

This makes sure that the timegraph is left in exactly the same state as when this resolution or factoring attempt was started.

7. For factoring, steps 3 through 6 are repeated, reversing the roles of the two literals so that the other one is entered this time.

As unification is done with every combination of arguments and the process repeated, several possible actions may be produced from the two literals. If all the arguments are constants, just

[11] Resolution actions here also pass the substitution information back to the general reasoner.

one iteration is done as there are no substitutions to be made.

Some examples of generalized resolution and factoring:

Resolving *(x before y)* against *(e1 after e2)*
Unification: *x/e1, y/e2*
<=> *(e1 before e2)* against *(e1 after e2)*
<=> Entering *(e1 after e2)* into the timegraph
<=> Compare relations between *e1* and *e2*:
test *before* with graph relation *after*
<=> Resolvable with residue *(e1 equal e2)*
Unification: *x/e2, y/e1*
<=> *(e2 before e1)* against *(e1 after e2)*
<=> Entering *(e1 after e2)* into the timegraph
<=> Compare relations between *e1* and *e2*:
test *before* with graph relation *before*
<=> Not resolvable

Factoring *(x before y)* and *(e1 after e2)*
Unification: *x/e1, y/e2*
<=> *(e1 before e2)* | *(e1 after e2)*
<=> Entering *(e1 after e2)* into the timegraph
<=> Evaluating *(e1 before e2)* to unknown
<=> Not factorable
Unification: *x/e2, y/e1*
<=> *(e2 before e1)* | *(e1 after e2)*
<=> Entering *(e1 after e2)* into the timegraph
<=> Evaluating *(e2 before e1)* to true
<=> Factorable with *(e2 before e1)* as the factor.

Chapter 4

The Specialist Interface

Having discussed how the temporal specialist can assist a resolution-based theorem prover, we now need a general interface that will allow the theorem prover to invoke the temporal and other specialists when appropriate. This interface should be general enough to handle any specialist we might imagine, and efficient enough that its cost is modest in comparison with the savings made possible by the specialists. We start with an effective theorem prover and several specialists, and describe the modifications needed to each as well as the specialist interface itself.

The resolution based theorem prover used by **ECoNet** has been under development at the University of Alberta for several years, most recently with de Haan [HaS86, Haa86]. The theorem prover uses a semantic net representation, and features automatic classification of entered clauses and organization in a topic hierarchy. Subnets for mental worlds organize modal expressions into separate areas. The inference method used is resolution, enhanced by topical retrieval of clauses to resolve against the problem clauses. Inference is also accelerated by a concept hierarchy that enables type inheritance.

The interface presented here enables the temporal specialist and other specialists to accelerate derivations attempted by the theorem prover. This required modifications both to the theorem prover and the specialists themselves. The architecture of the resulting hybrid is shown in Figure 4.1. The hierarchy specialist handles the concept and topic hierarchies, and since it is more deeply embedded into the system than the other specialists, it is shown inside the theorem prover itself. It uses the specialist interface for resolution and factoring, however. The temporal specialist has been described already in Chapter 2. The color specialist is very simple and determines disjointedness of color predicates. The number specialist and the set/list specialist will be described shortly.

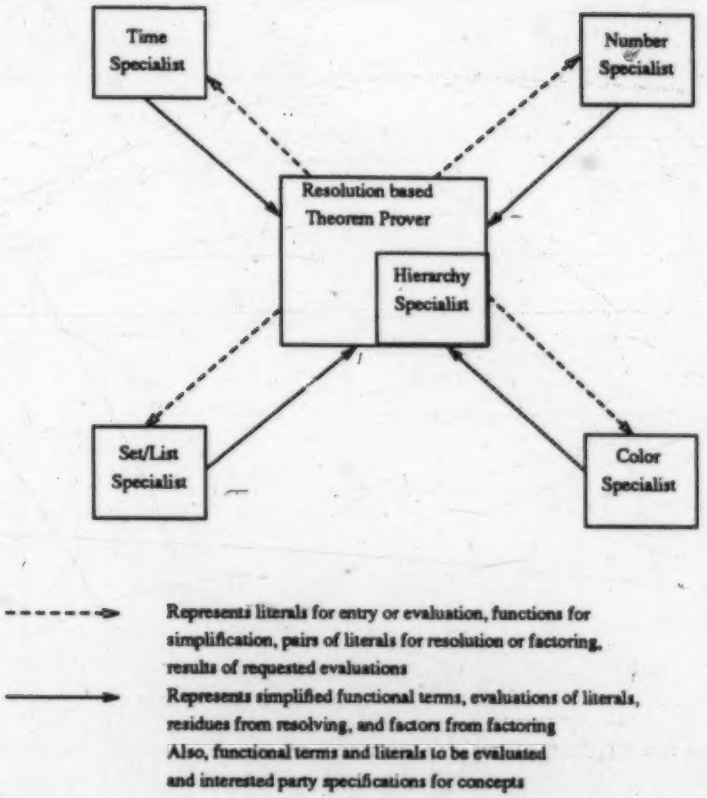
Time Specialist
Number Specialist
Resolution based Theorem Prover
Hierarchy Specialist
Set/List Specialist
Color Specialist
Represents literals for entry or evaluation, functions for simplification, pairs of literals for resolution or factoring, results of requested evaluations
Represents simplified functional terms, evaluations of literals, residues from resolving, and factors from factoring
Also, functional terms and literals to be evaluated and interested party specifications for concepts

Figure 4.1 Architecture of ECoNet

4.1. Changes Required to the General System[12]

A few changes were required to the theorem prover to enable the interface to operate. The expressiveness of the theorem prover needed extension, via *sorts* to differentiate kinds of concepts (a special kind of types), and *quoted expressions*, which can represent symbolic expressions and numbers. In addition, calls to the specialist interface had to be inserted. The notion of resolvability had to be extended to include residues and evaluation information provided by the specialists.

[12] Most of the changes actually required to ECoNet were done by John de Haan while re-implementing his thesis work in Lucid Common Lisp.

4.1.1. Sorts

In order for the specialist interface to be able to decide when a specialist is appropriate for some problem, the world of arguments must be divided so that events are different from physical objects, which are different from numbers, and so on. This was done by allowing a "sort tag" to accompany an argument. This is similar to typing a concept, except that *sorts* do not have fuzzy boundaries. Either something has been asserted to be an episode (of sort *episode*), or it is not an episode. Possible sorts include: *physical object, event/episode, time, number, symbolic expression* and *set*. They are expressed in the syntax by following an argument with an underscore and the sort. Types in this system, on the other hand, are unary predicates. For example,

e1_episode
(date '1987 '4 '1 '0 '0 '0)_time

Sorts are a special case of types that are recognized by the specialists.

Sorts are required since some predicates and functions overlap several specialist domains (for example, the predicate *equal*, and the function *relation* - in this implementation one can ask for the relation between two numbers, or between two temporal entities). In that case, we need some way of determining which specialist it belongs to (or specialists). Otherwise we will waste time and space saving and doing inference on arguments that have nothing to do with a particular specialist. For example,

[e1 equal e2]

could invoke both the time specialist and the number specialist, when *e1* and *e2* are episodes. Another option would be to have unique predicates and functions for a specialist (for example, we would then have functions *number-relation*, and *temporal-relation*). Then the predicate alone would be enough to indicate a specialist. However, this would not be sufficient to handle *equal*, unless we have a different *equal* for each specialist, which would lead to other problems. In addition, some specialists may work with several sorts of arguments and use this information to do different things (for example, the temporal specialist handles time and episode arguments slightly

differently).

Once a term is asserted to be of a particular sort, it cannot be changed, and remains of that sort until the end of the session. No inference is done with the sorts themselves - one cannot ask if *e1* is of sort *episode* (since *episode* is not a predicate, like types are). While the above sorts reflect the specialists currently in the system and those we envision as being useful, there is nothing to restrict the possibility of having, for example, a sort *wolf*, and a wolf specialist. Here the system would treat sort *wolf* differently from type *wolf* (*[W_wolf ...]* versus *[W wolf]*), as the latter is related to the type hierarchy, and the former is treated independently of any other sorts or types in the system. The sorts are a tool that allow specialists to quickly determine when something is in their domain, and what to do with it. An alternative method would have been to devise specialized naming conventions for terms which would involve specialists. Besides being clumsy, it would be much slower to parse each term than to simply check the sort.

Incomplete sort information is not considered in this implementation, as this system is at a low level and there will be other levels above it in ECoSystem that ensure that the proper sort information is included. If the sort of an argument has not been specified, the system makes no assumptions about what that sort might be, given the rest of the literal. Unless all arguments have an acceptible sort, a specialist will not operate on a literal or function.

Other uses of the term *sort* can be found in the literature. Sorts to de Champeaux [Cha78] are an extension of types, with non-empty intersections. Walther's sorts [Wal83] appear to correspond directly to type hierarchies. There are others as well, too numerous to mention.

4.1.2. Quoted Expressions

Some terms, such as numbers, symbolic expressions, and the values of some functional expressions are not easily expressed as semantic net concepts, or are too numerous to represent that way. For example, we do not want an individual concept for every possible number! To avoid this problem, another type of argument was developed, called a *quoted expression*. A quoted expression is of the form *'item* or *'(list)*. The first element of a list may be interpreted as a sort by some specialists. For example, a quoted expression representing a date has *time* as its first element, which the time specialist looks for. There are no limitations on the contents of such an object. They are treated as constants by the main theorem prover. For example, the functional expression *(date '1987 '4 '1 '0 '0 '0)* evaluates to a quoted expression

'(time 1987 4 1 0 0 0),

which is an absolute time representation recognized by the time specialist. The number 3 would be represented[13] as *'3*.

4.1.3. Unification

The sorts and quoted expressions required some changes in the unification algorithm. Two arguments may be unified only if they have the same sort, or one has an unspecified sort (in addition to the usual restrictions, of course). With the addition of specialists that can determine resolvability with non-identical predicates, unification does not always have to use a fixed order of arguments (first argument of literal one unified with first of literal two, and so on). The specialists decide what order the arguments should be unified in when resolving or factoring, and invoke the unification process themselves. During this process, there may be attempts to unify two constants. Normally, unless the two constants are identical, the unification attempt will fail at this point. However, the specialist may know some useful relationship between the constants, so that

[13] Although it would be nice to follow the Lisp convention that numbers are self-quoting, implementation considerations required that they be quoted here.

they don't have to be identical in order for a resolving or factoring action to be determined. In such circumstances, we would like the unification algorithm to return that it succeeded. To enable this, the specialists supply a decision routine to the unification algorithm to use when comparing two constants. No substitution of constants for constants is done however. This feature only serves to indicate that a unification attempt succeeds in some cases where it otherwise would fail. Logically, the effect is the same as if some axioms in the specialist's theory had been applied along with previous assertions. For example, if resolving *[a after x]* against *[b before c]*, where *[a before b]* has been asserted already, a unification attempt with c/x can succeed, by considering *[b before c]*, *[a before b]* and the transitivity axiom. This gives *[a before c]*, which is then unifiable with *[a after x]*.

4.1.4. Interface

As Schubert et al. [SPT87] suggest, there are several times during the operation of the theorem prover where specialists may be usefully invoked. These are literal evaluation, generalized resolving (determining whether two literals are resolvable and how), and generalized factoring (determining whether two literals are factorable and how). As well, assertions must be given to the specialists to maintain in their own representations, so there must also be an entry interface. Function evaluation is another case where specialists may prove useful. In this implementation, all functional terms are evaluated during simplification, which is done in the specialist interface, so no additional calls had to be added to the main theorem prover.

Literal evaluation takes place during simplification of assertions and newly entered resolvents, and requires only one interface call - during the evaluation phase of the theorem prover. The theorem prover normally checks the literal to see if it has previously been asserted (in the same form). If this fails, there may be another possibility for evaluation by some specialist, so the literal is sent to the specialist interface.

The specialists can be used to determine whether two literals are resolvable. If traditional methods to determine resolvability fail (identical predicates with different signs; unifiable arguments), the specialist interface is called. A specialist may be able to find several ways of resolving them (via different substitutions), and can place these on the resolving agenda for future consideration. Similarly, if traditional methods to determine factorability fail, the specialist interface is called. As with resolution, there may be many different unifications that can result in a factoring action. Examples of generalized resolving and factoring are found in the discussion of the temporal specialist in Chapter 2, as well as in the system examples in Chapter 5.

When a proposition is asserted, the appropriate specialists must be called so that they may enter the new assertion in their own representations. This is required for later inferencing. After an entered proposition has been normalized and simplified, the specialist interface is called to invoke the specialists that might be interested.

4.1.5. Resolution

The resolution theorem prover had to extend its notion of resolvability. If two literals are resolvable, traditionally the residue is null, but now that the specialists have been added, it is possible to get a non-null residue (from Stickel's partial theory resolution - mentioned in Chapter 3). For example, one possible resolving action from the literals $[x \leq y]$ and $[a \geq b]$ involves the residue $[a = b]$. The theorem prover had to take this into consideration when building the resolvent.

Also, when attempting to resolve, specialists may provide more than just substitutions and residues - they may also supply the evaluations of the two literals after substitution. In resolution, we are attempting to simplify set-of-support clauses. With this general goal in mind, we can now look at the possibilities when one of the literals evaluates to something other than unknown. If we start by trying to resolve literal l_i in the set-of-support clause

$$C_1 = l_1 / l_2 / \ldots / l_n$$

against literal k_j in the clause

$$C_2 = k_1 / k_2 / \ldots / k_m,$$

and literal l'_i (where ' means after substitution) evaluates to something other than unknown, we can simplify clause C'_1 immediately. If l'_i evaluates to *false*, we can infer the rest of the literals in C'_1, giving

$$C'_1 = l_1 / l_2 / \ldots / l_{i-1} / l_{i+1} / \ldots / l_n$$

If furthermore, k'_j evaluates to *true*, we can infer k'_j. If l'_i evaluates to *true*, we can infer that literal alone

$$C'_1 = l_i$$

If in addition k'_j evaluates to *false*, we have incompatible literals, and can infer the rest of C'_2,

$$k'_1 / k'_2 / \ldots / k'_{j-1} / k'_{j+1} / \ldots k'_m.$$

These evaluations can be easily calculated by the specialists, and give resolution more opportunity to simplify the set-of-support clauses. Note that only actions which simplify the set-of-support clause are considered - those which simplify the other clause do not necessarily get us closer to the goal of the null clause.

4.2. Specialist Requirements

A specialist must have a well-defined domain of predicates, functions, and possibly sorts and efficient methods for inference in that domain. Any inference made by a specialist must be sound. There is no requirement that the specialist be complete, as the general method can fill in any inference gaps, albeit less efficiently. The specialist must be able to evaluate both negative and positive literals if it does evaluation. That is, if it can evaluate the positive form of a literal, it must be able to evaluate the negation of the literal as well. Any tasks the specialists take on must be decidable, and quickly decidable. The specialists are to be used to accelerate the system, and so they must be fast, and they must ALWAYS return an answer (*unknown* is an acceptible

answer).

Specialists that maintain their own representation of some of the facts in the semantic net must be able to arrange them according to subnet to prevent inconsistencies. Also, all predicates associated with the specialist must be under the same topic node in the topic hierarchy; otherwise related propositions will never be found to resolve on (with the methods used by this particular resolution theorem prover).

At the highest level of the specialist, it must be able to understand, at least syntactically, predicate calculus. Usually this will quickly be converted to some internal language that the specialist actually uses. The interpretation of all functions and predicates used by the specialists must be consistent. That is, they must all "speak" the same language. For example, the function *max-of* means the maximum value of any of its arguments, whether or not that particular specialist is able to evaluate it. This requirement enables specialist communication, which will be described shortly. This does not necessarily mean that only one specialist will be able to evaluate a function or a literal, just that if more than one use the same function or predicate, the meanings are consistent.

4.3. The Specialist Interface

As mentioned earlier, there are several interfaces: for literal assertion and evaluation, function evaluation, and generalized resolution and factoring. Entry and evaluation, which each only involve one literal, merely require determining the specialists interested in that literal. Generalized resolution and factoring involve two literals, and require that specialists interested in both literals be invoked. To get quick and easy access to the specialists that may be useful, they are kept on the predicate's property list. The same is done for functions.

Not only are the appropriate specialists invoked with the literal or literals in question, they are also given an effort value to use. For evaluation, generalized resolution, or generalized factor-

ing, this effort level indicates how hard the specialist should work to get an answer. The temporal specialist uses an effort level to determine whether to do constant-time only operations (effort level 0), or full graph searches (any effort level greater than 0). For entry (which uses a different effort value), the effort level indicates how hard the specialist should work in determining consistency of input.

4.3.1. Function Evaluation

Functional terms are evaluated in a manner similar to Shostak's method of simplification, where applicable specialists are called to simplify the arguments of the function, recursively, until the function itself can be evaluated. A function is not evaluated until all its arguments have been. Then the list of applicable specialists is obtained from the function's property list, and called one after the other, until one returns an answer other than *unknown* or *nil*. For example

(date (add '1987 '1) '4 '1 '12 '0 '0)

would first use the number specialist to calculate the year argument, and then the temporal specialist to calculate the absolute time with all the arguments. Only the function name is used to determine which specialist to call; it is up to the individual specialists to check the arguments. Although the sort of the resulting concept or quoted expression will be one of the sorts used by that specialist, there is no requirement that the function's arguments have those sorts. For this reason, argument sorts cannot be checked by the interface and must be checked by the specialist itself. For example, the *date* functions takes six *number* arguments and returns a result of sort *time*. The temporal specialist normally works with sorts *episode* and *time*, so the numbers would not be recognized as being in its domain. If we add number to its list of sorts, we could get literals not in its domain being given to it (such as *[n_number equal '3]*).

4.3.2. Literal Entry and Evaluation

The mechanism for deciding which specialists should be called is just a cursory check to find specialists that are likely to be interested. Since much of the time no specialist will be involved, we do not want to waste too much time deciding which, if any to call. A list of potentially interested specialists is obtained from the predicate of the literal. For each specialist on the list that we are tempted to call, the list of its allowed first argument sorts is checked to see if any match the current literal's first argument (note: if the allowed first argument sorts are left unspecified, anything will match; but if specified, there must be an exact match). If so, the specialist is called, and can do the final checking of other argument sorts, whether the arguments are all constants, and so on, itself. Only the first argument is checked since there is no requirement that all arguments of a literal destined for a specialist be of the same sort. For example, for the temporal specialist, *[t1_time at-most-before t2_time '600000]* has mixed arguments sorts, and the last argument is not a temporal entity. However, a literal for a specialist almost always involves at least one term which is an entity specific to that specialist (like temporal entities for temporal literals), so here we have restricted the predicate-argument patterns to have this entity as the first argument in a literal. Some specialists have no such requirements (for example, the set specialist, which also handles equality, will take arguments of any sort for the predicate *equal* - the first doesn't have to be a *set*). These are handled by having an empty set of allowed sorts for that first argument. Another possiblility was to store complete sets of patterns for all argument-predicate combinations a specialist deals with. Besides requiring significant storage, checking the pattern by the interface could be quite slow. The specialists can check the other arguments faster than the inferface can.

On entry, all applicable specialists are called, as any one might later be required to use the information in inferencing. For literal evaluation, each specialist is called, one at a time, until an answer other than unknown is returned. For example, asserting

[t1_time before (date '1987 '4 '1 '0 '0 '0)]

would cause the time specialist to be invoked to enter it. Asking

?[e3_episode after junk]

would invoke the time specialist to evaluate it, which would reject it since the second argument is not an episode or time point (sorts *episode* or *time*).

4.3.3. Generalized Resolution and Factoring

When two literals are being tested for resolvability or factorability, the specialists associated with the predicate of each literal are obtained from their property lists, and the intersection of these two lists is our list of applicable specialists. No checking is done on arguments yet, as unification has not taken place (and cannot until the specialist decides how it is to be done), so the arguments are not known. Final checks on whether the resulting literals (after unification and substitution) are in a specialist's domain must be done by the specialist. For example, when resolving

[t1_time before t2_time] vs [x after t3_time],

both predicates have the temporal specialist on their property lists, and so that specialist would be called to try to resolve them. On the other hand, factoring

[t4_time after e3_episode] | [Wolf grey]

would not invoke any specialists as there are none that apply to both literals.

All applicable specialists are called for both generalized resolving and factoring. They may each add different resolving or factoring items to the agenda. Each may add several items, as the unifications may be done in different orders and still result in an action to be taken.

A specialist may do all the unifications necessary and actually determine the set of residues for resolving, or the factor(s) for factoring, and return these with the substitutions to the main theorem prover. Alternatively, it may merely return the relationship between the two predicates

and let the theorem prover handle the rest. The hierarchical specialist (the type hierarchy included with the theorem prover), and the color specialist just return this relationship. If the two literals are both positive, and their predicate relationship is disjoint, they may be resolved with a null residue. Similarly, if one predicate subsumes the other when factoring, the main system could determine which was the factor. This allows specialists to be as simple as possible. The color specialist and hierarchical specialist do not require control over which arguments are unified with which, as the predicates they deal with are one-place. The contribution they make is the determination of the predicate relationship. Once this is known, standard techniques can be used, and there is no point is repeating them for each specialist. The temporal specialist, on the other hand, can generate numerous different resolving actions by changing the order of argument unification, and testing each possibility, so it can contribute much more useful information to the main theorem prover.

4.4. Communication between Specialists

Although all arguments are evaluated during entry and evaluation before invoking the specialist, a specialist may still need additional information to complete its task. For example, if the temporal specialist is trying to evaluate whether an event *e1* with upper absolute time bound *(1987 mm 01 00 00 00)* is before *(1987 06 01 00 00 00)*, it will require information on the bounds of *mm* (if any are available). Similarly, a naive or qualitative physics specialist might need to know if the event of the robot hand going across the table happened before or after the vase of flowers was placed there in order to determine the current position of the vase. To get this extra information, specialists can request the evaluation of functional terms or literals. The specialist interface then decides which specialist will do the evaluation, as described above. Since there is no way to enforce ordering of input, there may be times where a specialist wants something evaluated for which there is no information yet. When (or if) that information is asserted, the specialist needs to be notified.

Two types of communication have been implemented for the specialists: *immediate evaluation*, where a specialist asks for the evaluation of a particular functional term or literal, and *delayed communication*, where the specialist is notified that something of interest in another domain has been asserted. For immediate evaluation, the functional term or literal is sent out from the specialist for evaluation, and the result (a concept or quoted expression for a functional term; *yes*, *no*, or *unknown* for a literal), is sent back to it. For delayed communication, the specialist communicates to the the interface the particular concept it wants "watched", and a literal to reassert when something involving that concept is asserted. The specialist receives a request to assert that literal for any assertion involving the concept.

4.4.1. Additional Specialists

Temporal inference is certainly not the only domain which has potentially explosive inferencing. Reasoning with the ordering relationships between numbers, as well as determining set membership are also problems which we will have to deal with, even for understanding ordinary discourse. For example, if we are told that eleven of twelve jurors agreed on a guilty verdict, we should be able to figure out that exactly one juror demurred. Similarly, if we are told that John is a member of the curriculum committee, and all members of the curriculum committee are members of the faculty council, we should be able to figure out that John is a member of the faculty council. To accelerate reasoning in these areas, some additional specialists are required - a number specialist, and a set specialist. These specialists currently handle function evaluation, literal evaluation, and literal assertion. Neither the number or the set specialist currently does any generalized resolving or factoring although plans are underway to implement this. In addition, specialists which handle those domains also interact with the temporal specialist, so we can use them to test the communication between specialists.

The number specialist deals with orderings and functions on integers and reals, and named integer or real constants. It handles the relational predicates *less-than* or *lt* (<), *greater-than* or *gt*

(>), *lt=* (≤), *gt=* (≥), and *equal*, each of which can take two arguments of sort *number*. The functions *max-of*, *min-of*, and *add* are also part of its domain. Another function, *relation* returns the predicate relation between the two number arguments given to it. All arguments to the functions must also be of sort *number*. The number specialist interacts with the temporal specialist in absolute time and duration specifications, especially if they are incomplete and later filled in. For example, assertion of

[e1_episode before (date '1987 mm_number '01 '12 '00 '00)]

would require that the time specialist ask the specialist interface to evaluate *(max-of mm)*[14], which invokes the number specialist. Another example of interaction was noted earlier in the description of function evaluation.

The set specialist handles set membership, and some simple operations on sets, as well as a limited version of equality (quickly decidable equalities only). Its domain consists of two predicates: *equal* and *member-of*, and one function *set-of*. *Set-of* takes all of its arguments (which may be of any sort), and constructs a quoted expression recognized as a set. *Equal* can take two arguments of any sort. Currently any equality may be tested, but only equalities between sets are stored. Named sets can be assigned members by asserting something like

[s1_set equal (set-of member1 member2 ...)].

The predicate *member-of* takes any entity as its first argument, but the second must be of sort *set*. This specialist overlaps the temporal specialist with the predicate *equal*. For example,

?[e1_episode equal e1_episode]

could be answered by either the temporal or set specialist.

---

[14] We always want the most complete, but correct information available. Since *mm* can never get bigger than its maximum, this is the safest bound to use on assertion. However, if we were trying to evaluate this literal, we would use the minimum, since if *e1* were less than the minimum of that time, we can be assured that it will be less than any other value that time might have.

4.4.2. Immediate Evaluation

Immediate evaluation takes place when a specialist wants a functional term or literal evaluated immediately. The item to be evaluated is passed to the specialist interface, which then decides who to send it to for evaluation, in exactly the same way as if it had come from the general theorem prover. Immediate evaluation may be used in any of the possible phases of a specialist (entry, evaluation, etc).

One example of this was shown above in the description of the number specialist. The temporal specialist may also ask for the *min-of* something, as it would if entering

[t1_time at-least-before t2_time d_number],

where it would ask for *(min-of d)*.

The current specialists do not require literal evaluation to do their jobs, but it is not inconceivable that a specialist might want to use it. The example mentioned earlier involving the naive or qualitative specialist would require communication via literal evaluation with the temporal specialist.

4.4.3. Delayed Communication

Sometimes an request for immediate evaluation will not be evaluable as the necessary information has not been asserted yet. In the example in the previous section, if nothing has been asserted about *d* yet, *(min-of d)* will not have a value. Here a specialist may indicate to the specialist interface that it wants to know if anything more is asserted about a concept, as functional or literal evaluation may then give a value, or a more constrained result. This is done by maintaining an *interested party* list for concepts. The interested party list contains information about the specialist that is interested in the concept, the current net environment (the subnet name), and the literal that gave rise to the specialist's interest. For example, asserting

[t1_time before (date '1987 mm_number '1 '0 '0 '0)]

would make the temporal specialist add an entry to *mm*'s interested party list. This entry would consist of the specialist (*time-specialist*), the subnet (*main*), and the literal given to the temporal specialist (after term evaluation) *[t1 before '(time 1987 mm 1 0 0 0)]*.

The interested party list associated with a particular concept is checked whenever a proposition involving that concept is asserted (after the specialists interested in the new literal have been invoked). Each literal on that interested party list is re-asserted to the specialist that put it there (as long as the old literal and the new refer to the same subnet). The specialist may then ask for immediate communication to get functional or literal evaluations with a useful result. There is no point asserting the new literal to these interested specialists because they won't know what it means. For example, in the above example, if

[mm ≤ '5]

were later asserted, the temporal specialist will have no idea what ≤ means, as it is not a temporal predicate (in this implementation we use *before* for that temporal relation). However, after the number specialist has seen that literal and updated *mm*'s maximum accordingly, the temporal specialist can ask for *(max-of mm)* and get a useful answer to use in its absolute time propagation. An example will be shown shortly. Later, *mm* can be tightened even more (to a maximum of 3, for example), and the temporal specialist will be notified so that it can update its absolute times.

The interested party communication is used for assertions only, as they will remain permanently in the net, and so any further assertions can enhance that information. Evaluations and generalized resolutions/factorings are fleeting and only remain for the duration of the question. If later some information is received that may have been useful in answering a previous question, it is too late to be of help there, but assertion information may still be useful for future questions. During a resolution or factoring attempt, items may get added to an interested party list by the entry phase of the attempt; however, they are retracted at the end of the attempt along with the changes to the temporal representation.

Another possibility for handling the interested party list was to keep track of what the specialist wanted to know about the concept (e.g. *max-of*), and a location within the specialist to start at again. There were dangers inherent in this method however - it was clumsy and might not be general enough for all possible specialist interactions (for example - literal evaluation). Reasserting the original literal gives the maximum generality while being the most simple and effective implementation.

Using this method also solved some potential problems for the temporal specialist during propagation. When propagating absolute times, the durations are taken into effect and added to the absolute time being propagated, or subtracted from it, as necessary. If the absolute time is not completely specified, this can be messy. For example, note what happens if we assert

[t1_time before (date '1987 mm_number '1 '0 '0 '0)]

followed by

[t2_time exactly-before t1_time '6000000]

Normally the temporal specialist tries to calculate the maximum absolute time it should propagate using the given time and taking the duration into account. Here it would want to give *t2* an absolute time of *(1987 (subtract mm 3) 20 13 20 0)*. However, this cannot be represented or handled by the temporal specialist. One possibility is to introduce another named constant, say *mm2*, and assert that *[mm2 = (subtract mm '3)]*. This really gives no more information than the original absolute time. We already know *t2* is before *t1* from the timegraph itself, so it is not helping us there. In an absolute time comparison, no more can be gained by comparing a number with an unknown month field *mm2* than with an unknown *mm*. But when

[mm ≤ '5]

is asserted, the information necessary to enable the propagation using the durations is available. By reasserting the literal originally asserted to the time specialist

[t1 before '(time 1987 mm 1 0 0 0)],

we force the time specialist to treat this as a new assertion and to start from the beginning. This

ensures that we won't miss any useful inferences that might be made as a result of that assertion, and doesn't require keeping hooks to "where we left off" in the specialist. The temporal specialist can now ask for *(max-of mm)*, and get a useful result, 5, which gives it a completely specified absolute time to use in propagation. Now *t2* will get an absolute time maximum of *(1987 2 20 13 20 0)*.

Chapter 5

Experimental Results

5.1. Examples

Interesting deductions we can request involve "mixed" inference (i.e. involving both temporal relations and others). For example, in the story of Little Red Riding Hood, deciding whether the wolf was alive when everyone was eating the goodies in the basket requires using knowledge that one cannot be alive after one is killed, and temporal inference to determine that the episode of eating the goodies came after the episode of killing the wolf. Other examples of the uses of such "mixed" reasoning in planning and problem-solving can be found in [AlK85]. Figure 5.1 shows an example of the system in operation, trying to answer a question about the story of Little Red Riding Hood, similar to the one mentioned earlier. The question the system tries to answer is

Was the wolf alive at some time after everyone talked or after everyone ate the goodies?[15]

which we translate to

?[E x_episode [W alive x] & [[x after all-talk] | [x after-1 (end-of eat-goodies)]]][16].

[15] Although somewhat awkward, this question illustrates all the areas of temporal specialist assistance without being overwhelming.

[16] The question is phrased using *after-1*, and specifying *end-of* to show that the system can actually handle predicates of different strictness, and combinations of time points and events. The questions could just as easily have been phrased as *[E x_episode [W alive x] & [[x after all-talk] | [x after eat-goodies]]]*.Also note that a sort tag (*_episode*) is attached to some terms; this is used to differentiate terms so that the temporal specialist is called only when appropriate.

```
==> ?(E x_episode (W alive x) & ((x after all-talk) or (x after-1 (end-of eat-goodies))))
Entering disproof clauses:
(W ALIVE SCON-380) (depth 1)
((SCON-380 AFTER ALL-TALK) | (SCON-380 AFTER-1 (END-OF EAT-GOODIES))) (depth 1)

  Time Specialist: END-OF (EAT-GOODIES) evaluated to EAT-GOODIESEND
((SCON-380 AFTER ALL-TALK) | (SCON-380 AFTER-1 EAT-GOODIESEND)) (depth 1)

  Time Specialist: Trying to factor (SCON-380 AFTER ALL-TALK) and (SCON-380 AFTER-1 EAT-GOODIESEND)
  Time Specialist: Factored to (SCON-380 AFTER ALL-TALK)
(SCON-380 AFTER ALL-TALK) (depth 1)

  Resolved (W ALIVE SCON-380) in the disproof clause
    (W ALIVE SCON-380)
    against (~ U-VAR-2 ALIVE EPISODE-VAR-1) in ((~ U-VAR-1 KILL U-VAR-2 EPISODE-VAR-2)
      | (~ EPISODE-VAR-1 AFTER EPISODE-VAR-2) | (~ U-VAR-2 ALIVE EPISODE-VAR-1))
      yielding ...
((~ SCON-380 AFTER EPISODE-VAR-1) | (~ U-VAR-1 KILL W EPISODE-VAR-1)) (depth 2)

  Resolved (~ U-VAR-1 KILL W EPISODE-VAR-1) in the disproof clause
      ((~ SCON-380 AFTER EPISODE-VAR-1) | (~ U-VAR-1 KILL W EPISODE-VAR-1))
      against (WOODCUTTER KILL W WOLF-DEMISE) in (WOODCUTTER KILL W WOLF-DEMISE)
      yielding ...
(~ SCON-380 AFTER WOLF-DEMISE) (depth 3)

  Time Specialist: Trying to resolve (~ SCON-380 AFTER WOLF-DEMISE) against (SCON-380 AFTER ALL-TALK)
  Time Specialist: Resolved with residues null
  Resolved (~ SCON-380 AFTER WOLF-DEMISE) in the disproof clause
      (~ SCON-380 AFTER WOLF-DEMISE)
      against (SCON-380 AFTER ALL-TALK) in (SCON-380 AFTER ALL-TALK)
    yielding the null clause.
NO
```

Figure 5.1 Example of ECoNet in operation[17]

In this translation, it is taken for granted that there is a referent determination process for certain time adverbials that has selected the episode called *all-talk* as the referent of *everyone talked*, and *eat-goodies* for *everyone ate the goodies*. Similarly, *W* is the individual referred to as *the wolf*. It

[17] This example shows actual output of the system, edited for clarity and brevity. Bold print is user input; the rest, system output. The existentially quantified variable *x* in the question has been converted to the skolem constant *SCON-380* used in the proof. The timegraph contained, among other relations, that *all-talk* is during *eat-goodies*, and that *eat-goodies* is after *wolf-demise*, the episode corresponding to the woodcutter killing the wolf.

is sometimes possible to handle this with existential quantification (e.g. *[E x_episode [everyone eat goodies x] & [x ...]]*), but this only works satisfactorily when a *yes* answer is expected. If a *no* answer is expected, the system currently answers *unknown*, as there may be events fitting that description it doesn't know about (default reasoning would be required otherwise).

The example in Figure 5.1 required about 57.5 seconds on a Sun 3/75 - just under a minute. This was with a knowledge base consisting of about 150 propositions (which are normalized into over 450 clauses), some for general knowledge and some for the simplified version of the the Little Red Riding Hood story we use for testing (it consists of over 70 propositions, 20 of which temporally relate episodes in the story). The concept and topic hierarchies, and general and specific knowledge for the story are in Appendices A3 through A6, respectively. The number of steps required for the proof was small, although the operations accelerated by the specialist would normally have required many applications of temporal axioms.

The next example shows the communication between specialists - both immediate and delayed. A temporal assertion is made with an incompletely specified absolute time that is specified by a later assertion captured by the number specialist. Note that *mm* has a literal added to its interested party list, and that the unspecified absolute times are propagated without including the durations. Later, after *mm* is asserted as having a maximum of 5, note that the first literal is re-asserted from its interested party list, and that the time specialist now has completely specified absolute times and can subtract durations to get more precise bounds. Immediate communication is done when the time specialist asks for an evaluation of *(max-of mm)*. An example of delayed communication is the re-assertion of the literal on *mm*'s interested party list.

```
==> (e1_event before (date '1988 mm_number '1 '0 '0 '0))
  Specialists: Evaluating function DATE ((QUOTE 1988) MM (QUOTE 1) (QUOTE 0) (QUOTE 0) (QUOTE 0))
  Time specialist: DATE evaluated to (QUOTE (TIME 1988 MM 1 0 0 0))
  Specialists: Entering (E1 BEFORE (QUOTE (TIME 1988 MM 1 0 0 0)))
  Specialist interface: Adding to interested party list of MM
      TIME-SPECIALIST / (E1 BEFORE (QUOTE (TIME 1988 MM 1 0 0 0)))
  Specialists: Evaluating function MAX-OF (MM)
  Number specialist: MAX-OF (MM) evaluated to
  Time Specialist: Adding absolute time maximum (1988 MM 1 0 0 0) to E1END
  Time Specialist: Adding absolute time maximum (1988 MM 1 0 0 0) to E1START
==> (e2_event exactly-before e1_event '6000000)
  Specialists: Entering (E2 EXACTLY-BEFORE E1 (QUOTE 6000000))
  Time Specialist: Adding absolute time maximum (1988 MM 1 0 0 0) to E2END
  Time Specialist: Adding absolute time maximum (1988 MM 1 0 0 0) to E2START
  Time Specialist: Adding duration minimum 6000000 between E2END and E1START
  Time Specialist: Adding duration maximum 6000000 between E2END and E1START
==> (mm lt= '5)
  Specialists: Entering (MM LT= (QUOTE 5))
  Number specialist: Entering MM LT= (QUOTE 5)
  Number specialist: Setting maximum of MM to 5
  Specialist interface: Reasserting from interested party list of MM
      TIME-SPECIALIST / (E1 BEFORE (QUOTE (TIME 1988 MM 1 0 0 0)))
  Specialists: Evaluating function MAX-OF (MM)
  Number specialist: MAX-OF (MM) evaluated to (QUOTE 5)
  Time Specialist: Adding absolute time maximum (1988 5 1 0 0 0) to E1END
  Time Specialist: Propagating absolute time maximum (1988 5 1 0 0 0) to E1START
  Time Specialist: Propagating absolute time maximum (1988 2 20 13 20 0) to E2END
  Time Specialist: Propagating absolute time maximum (1988 2 20 13 20 0) to E2START
```

Figure 5.2 Example showing Communication between Specialists[18]

A larger test base of questions has been set up using the same database of Little Red Riding Hood information used in the previous examples. It consists of some purely temporal questions (such as *Did the episode named wolf-eat-gm occur before the episode named wolf-eat-lrrh?*, some mixed questions (such as the one above about the wolf being alive at a particular time), and some involving no temporal inference at all (for example, *Is Little Red Riding Hood a rock?* or *Does Little Red Riding Hood like to eat cake?*). Appendix A7 contains these test questions and the answers the system returns for them. The time required for an answer for each question

[18] User input is in bold print; system output in regular print.

varied greatly, from a few seconds for the purely temporal questions (which involve simple literal evaluations) and some of the simpler questions involving no temporal inference, to a few minutes for the more complicated ones involving "mixed" inference, and quantification.

5.2. Positive Results

The combined **ECoNet** system can solve some problems that were not feasible to solve before the addition of specialists. These problems either required so many steps in resolution that they were not practical to run, or involved so many axioms that it was not feasible to add them. Both were true for temporal problems. Because temporal orderings are transitive, it is easy to get a combinatorial explosion of steps in trying to infer *a before b* from some sequence *a before c before e before ... before b*. Because of the number of predicates, especially with strictness values, the number of axioms potentially involved is huge.

Each specialist only has to concentrate on its own domain, and not worry about possible interactions with other specialists, so the specialists themselves may be quite simple. A specialist may divide its problem up into subproblems (literal evaluations, function evaluations, and even residues in generalized resolution) without having to consider which other specialist can solve these subproblems, or even if there is one that can. Decisions about what representation and methods to use in implementing each specialist are driven by the needs of the specialist, not any artificial constraints placed on the system by choice of representation and/or methods. Each specialist may be fine-tuned independently, without considering interactions with the rest of the system.

New specialists may be added easily without disrupting the system. To add a new specialist, the specialist itself (some Lisp code) is required, and the list of predicates and functions it deals with (so the specialist can be added to the list of interested specialists for those predicates and functions). The predicates must be placed under the same topic in the topic hierarchy. No

changes are required to the general inferencer, the interface, or any of the other specialists.

5.3. Negative Results

It is harder to tell exactly what the system is doing as some inference is hidden in the specialists. Bugs and "holes" are easy to introduce into the system via the specialists. For the kinds of inference done by the specialists implemented here, these are not serious problems. The types of reasoning are well understood and in most cases almost obvious, so justification is not essential, and bugs are easy to detect and fix. "Holes" are not really a problem as long as the general inference mechanism can handle the inferences the specialists do, albeit less efficiently. However, at this stage, the language of the general theorem prover does not cover all the constructs handled by the specialists, so that the holes in the specialists can remain as holes in the entire logical mechanism.

The interface slows the system down slightly, but not significantly. However, the specialists themselves (especially the temporal specialist, which is fairly large) can slow the system down considerably, due to the extra complexity involved in determining resolvability or factorability. During experimentation, it was noted that the use of the timeframe argument, although resulting in more nearly optimal timegraphs, resulted in more time being taken for a proof than if the timeframe argument were not used. The resulting proof with this argument was actually shorter, but the time required to check the extra unifications possible was longer. This suggests rather strongly that it is a mistake to allow all the different versions of *before*, *during*, etc in the "deep" logical representation. Instead, we should just use some convenient minimal set, closely related to natural language (perhaps just *before* and *strictly-before*). The timegraph would still be able to efficiently represent all the relations necessary, but there would be fewer alternatives to check for possible resolving actions.

As with any theorem prover based on first order logic, this system is prone to computational

explosions. Although the specialists accelerate the system by short-cutting some long proofs, they also give the system more possible resolving actions than a typical resolution-based theorem prover.

5.4. Comparison with Other Systems

As already mentioned, this system belongs to the class of theorem-prover-based hybrids, which use specialists just to accelerate inference - not to change the overall expressiveness and reasoning power of the system (in principle). In this respect, it is similar to Krypton, Horne and LOGIN. Although literal and function evaluation appear to be done in many of the systems, most of them don't do generalized resolution or factoring. Horne, for example, only allows the specialist to assist in literal evaluation. Specialists in Shostak's system assist mainly in function evaluation. The Nelson-Oppen system allows specialists to do some literal evaluation and some function evaluation. Krypton, however, does allow its specialist to do generalized resolution. The residues calculated by Krypton can be quite complex, involving quantification. While ECoNet has no constraint on the complexity of residues that a specialist may create, the specialists used in this thesis generated quite simple residues, simplifying the resolving process. The communication between specialists is similar to the *equality propagation* of Nelson-Oppen's system, although in ECoNet, more than just equalities are communicated.

Many of the systems do not incorporate enough specialists to tell if their method is general or easily extendible (most have only one "specialist" - Horne, Krypton, KL-TWO), or severely restrict the types of specialists (Shostak's and Nelson-Oppen's must be equational) and what they can do (simple evaluations for most). ECoNet incorporates five specialists with few restrictions on them, and allows them to assist the main theorem prover in several ways.

Chapter 6

Conclusions and Further Work

6.1. Conclusions

Our thesis was to hypothesize a method for combining efficient special methods with a general purpose theorem prover in such a way that the efficiency of the special methods counteracts the combinatorial explosions usually associated with a general theorem prover. This would give an efficient system that still handles a large domain. Previous approaches limit the domain severely, aren't general enough for many kinds of specialists, aren't really that efficient or don't exploit the specialists capabilities as much as they could.

A general interface between specialists and the general theorem prover was developed that allows a specialist to accelerate the main inference mechanism through function evaluation, literal evaluation, and generalized resolution and factoring. To allow the maximum flexibility in the specialists, communication between them was also incorporated. This is accomplished by allowing immediate evaluation of functional terms and literals, as well as notifying specialists that an assertion of interest to them had been made, although not directly in their domain.

Several specialists were incorporated to test this interface, the major one being a temporal specialist. The temporal specialist was based on an efficient temporal model developed by Taugher and Schubert, and was extended to be more useful in this environment. Besides extending its reasoning power to handle both strict and non-strict ordering, additional inferencing capabilities were added for generalized resolving and factoring. In addition, a number/arithmetic specialist, a very simple color specialist, and a set/list specialist were incorporated. The temporal and number specialists communicate through absolute time and duration specifications. The combined system can do proofs in a few steps that normally would require many.

There is a trade-off between the development time for a specialist and the efficiency it adds

to the combined reasoner. The time spent in developing and enhancing the temporal specialist was well invested, since temporal reasoning is so essential in many AI applications. Although a specialist may slow down single steps of the system (because of the greater complexity of tests for resolvability or factorability), it can shorten the number of proof steps drastically. Some problems that previously were not feasible can now be done in reasonable time. One drawback in such a system is that some of the inferences are hidden within the specialist, and are thus "invisible". However, since the type of inference performed by the specialists (particularly the temporal and number specialists) is well understood and almost "obvious", justification is not essential - provided bugs and "holes" have been eliminated.

6.2. Extensions and Further Work

Several extensions and enhancements of ECoNet could be developed to make it an even more useful system, including adding more specialists, and enhancing existing ones. The general reasoner could be enhanced to do modal inference and default reasoning. The range of temporal inference handled by the system could be extended by adding axioms to deal with such things as days of the week. However, there is a limit to what we can get ECoNet to do by itself ; some things will have to be done at a level above it. For example, with regard to temporal inference deciding the duration lengths must be done before ECoNet is invoked. A duration of a day may be considered exactly 24 hours, a range of 23 to 25 hours, or even a range of 16 to 32 hours, depending on context.

6.2.1. Closure

Another possibility for improvement would be the addition of "closure" assumptions. That is, the system could assume that it knows all the relevant things (individuals and episodes) to fully answer a question. Then some questions that are now answered as "unknown" could be answered definitely as *yes* or *no*. For example, the system could assume that it knows about all episodes of a particular kind. If a question is asked about the existence of such an episode satisfying certain constraints and the only known episodes of that type violate the constraints, then the result would be *no*. Then we could use existential quantification for specific noun phrase referents, and get the answers we expect, instead of *unknown*. Consider, for example, the question: "Did the wolf eat grandmother after he ate Little Red Riding Hood?". We could phrase this question as

?[E x_ep E y_ep [W eat GM x] & [W eat LRRH y] & [x after y]]

or even

?[E x E y_ep E z_ep [x wolf] & [x eat GM y] & [x eat LRRH z] & [y after z]].

In this case, there is only one episode of the wolf eating grandmother, and only one of him eating Little Red Riding Hood, so we should be able to answer this as *no* immediately. But, as there is nothing to prevent two episodes of that type from existing in the questioned temporal order, the system currently answers *unknown*. During resolution, if a null resolvent or a clause evaluating to *false* is encountered, we are finished. If this is during the "proof" attempt, we answer *yes*; during the "disproof" attempt, *no*. For *true* evaluations, no special action is taken, and the resolution process continues. However, in circumstances like the above question, where there is only one possible unification for the variables, that *true* value does tell us something - in the proof attempt, we can answer *no*; the disproof *yes* (the reverse of what the *false* evaluation lets us do). In the above example, after several resolutions and unifications, we would have the following literal in the proof attempt:

[~wolf-eat-gm after wolf-eat-lrrh],

which evaluates to *true* with the assistance of the time specialist. Because we know that these are the only possible episodes of the required type, we should be able to stop and have the system answer *no*. Great care would have to be taken to ensure that this *unknown->no* or *unknown->yes* step is taken only where it is warranted by implicit assumptions about the completeness of certain kinds of knowledge. A *true* value will only be obtained by evaluating a literal with all variables filled in by unification. If there is only one unification, this is the only possible evaluation result we can get. If there are other unifications, one might yield a *false*, or even *unknown* value, so we must wait until all are done. One possibility is to save the fact that there was a *true* evaluation, and if at the end, the system wants to answer *unknown*, we could replace that by the appropriate answer, depending on whether the *true* came in the "proof" or "disproof" attempt. However, this will not work if some unifications evaluate to *unknown*. Also, it could be very slow waiting for the *unknown* result, and we should be able to stop immediately on getting *true* (in the right circumstances). The level requesting the question answer could set a flag indicating that it expects only one possible unification (i.e. at least one variable is definite). Perhaps a new syntax item *D* indicating definite quantification could be introduced, which would be equivalent to *E* (existential quantification) in all respects except that if any *D*s are detected in a question, this "stop at the first true" flag could be set. However, a clearer understanding of the logical basis of such inferences would be needed before implementing this - specifically, what "completeness assumptions" are being used, and how they are justified.

6.2.2. Other Theorem Provers

As Prolog uses a resolution-based theorem prover, some aspects of this method should work for extending it as well. Certainly function and literal evaluation should be fairly straightforward, although generalized resolving might require extensive revisions to locate potential resolving actions with the expanded notion of resolvability. For other kinds of theorem provers, more may be required of the specialists. Specialists in the resolution-based environment needed to be able to do simple evaluations, and apply two inference rules - resolution and factoring. Other theorem provers can handle numerous inference rules, which could lead to much more complexity in the specialists, if they were expected to handle all of the inference rules. Some other theorem provers can also handle non-canonical forms of input, but if we restrict the specialists to functional terms and literals, as we did for this system, no additional problems should result.

6.2.3. Specialists

We must continue searching for ways to improve the specialists, either by extending their capabilities or making them more efficient. For example, the color specialist used here is a simple one, and as there is a theory for a better color specialist [Pap82], this one could be improved. The set specialist, which also handles equality, could be improved to store equalities given to it for later use, maintaining a kind of congruence closure. This might require that another kind of communication be added to the system - assertional. Then if a specialist detected an equality while handling an assertion, it could report that equality back to the interface, where it would then be sent to the set specialist. The question here is: should the set specialist be responsible for all equalities, or should the main system be enhanced to handle equality itself? As our intention was to have a system which in principle could do all of this inference itself, using specialists only to accelerate inference, the latter seems the more reasonable approach.

To enhance the system even more, we could add more specialists. A spatial specialist

would handle spatial relationships, for instance the transitivity inherent in such predicates as "in". A naive or qualitative physics specialist, (for example [KlB85]) which would deal with motion and such things as deciding whether a bunch of roses will fit into a wine glass would prove most useful, both in extending the system's capabilities, and further testing the interactions between specialists. It would likely make extensive use of the other specialists (especially time, the spatial specialist, and possibly the number/arithmetic one).

Application-dependent specialists could also be added for specific uses of the system. For example, a specialist to shortcut reasoning about picnic baskets or wolves could shorten some proofs in the story of Little Red Riding Hood. For example, one question we might ask is

Is there anything in the picnic basket that Little Red Riding Hood likes to eat?.

The picnic basket specialist would contain information about the previous assertions about the picnic basket (that it contained cake, etc), as well as general information about picnic baskets (they always contain people food), so could potentially shortcut such a question. Or a specialist about cars could be added if the system were to be used for accident reports. Here the trade-off between the effort required to develop the specialist and the benefits gained from it is not quite so heavily weighted in favor of the specialist. In the system as described here, such application dependent information could be clustered (perhaps under subnets). Application dependent specialists would not be required then, as the general-purpose specialists could probably accelerate inferencing in their domains just as much as having an additional specialist. The system already answers the question given above, and does so in about 2 seconds.

6.2.4. Effort Levels

Currently, specialists on the property list of a predicate or function are not in any particular order. When more than one specialist is potentially able to solve the problem (evaluation of a literal or function, or determining resolvability or factorability), it would be more efficient if the specialist that required the least effort were the first called, and then one that requires more work, and so on up to the one that involves the most processing. With the specialists implemented in this thesis, there was not enough overlap to consider this, but if additional specialists with more overlap were incorporated, especially if any require significantly large processing times, we might want to optimize specialist selection.

Rather than using a fixed effort level for all specialists, the interface could decide for each problem it sees what the effort level should be. This would require some method for determining "how hard" the current problem is, and another to determine how hard we should try.

6.2.5. Parallelization

Another possibility for combining the specialists is to have each specialist running on a different processor (specialist *servers*). The interface would send the given problem to all specialists, and simply use the result of the first specialist to respond. This removes the need to check each literal for the appropriate specialists. However, this would waste processor power as specialists would be invoked that are not appropriate, as well as adding some complexity to the specialists as they would have to do the initial checking now done by the interface as well. One would probably still want the initial checking done before invoking these other processors. Using a parallel system like this would only prove useful if there was a great deal of overlap for some problems so that several specialists were applicable to solve the problem (similar to the reasons for having specialists ordered by effort level). Significant savings could result since instead of waiting for one specialist to finish, and checking to see if it got an answer, waiting for a second to

finish, and so on, we let them all go and take the first answer to return.

Parallelization could prove beneficial in other uses of the system as well. When interpreting natural language, there are usually numerous such question answering processes to be done. As each can take some time, and we want the shortest possible response time, it would be nice if at least some of the proofs could be done in parallel. This raises questions about whether the semantic net and all specialist information should be repeated on every machine being used, or whether there is some way to maintain a central semantic net and special representations, yet still allow different processes to use that information to solve different problems. As the current method uses the semantic net to store and classify its partial results, it would be difficult to ensure that the different proof attempts don't clash.

References

[AiN86] Hassan Ait-Kaci and Roger Nasr, Logic and Inheritance, *13th ACM ACM Symp. on Prin. of Programming Languages* , 1986, 219-228.

[All83] James F. Allen, Maintaining Knowledge about Temporal Intervals, *Communications of the ACM 26*, 2 (1983), 832-843.

[All84] James F. Allen, Towards a General Theory of Action and Time, *Artificial Intelligence 23*, 2 (1984), 123-154.

[AGF84] J.F. Allen, M. Giuliano and A.M. Frisch, The HORNE Reasoning System, Tech. Rep. 126, Computer Science Department, University of Rochester, Rochester, NY, 1984.

[AlK85] J.F. Allen and H.A. Kautz, A Model for Naive Temporal Reasoning, in *Formal Theories of the Commonsense World*, J.R. Hobbs and R.C. Moore (ed.), Ablex, Norwood, NJ, 1985, 251-268.

[BFL83] R.J. Brachman, R.E. Fikes and H.J. Levesque, Krypton: A Functional Approach to Knowledge Representation, *IEEE Computer 16*, 10 (1983), 67-73.

[BGL85] Ronald J. Brachman, Victoria Pigman Gilbert and Hector J. Levesque, An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON, *Proceedings of IJCAI-85 I*, (1985), 532-539.

[BrR85] Daniel C. Brotsky and Charles Rich, Issues in the Design of Hybrid Knowledge Representation and Reasoning Systems, *Proceedings of the Workshop in Theoretical Approaches to Natural Language Understanding*, Halifax, N.S., 1985.

[BBM82] Alan Bundy, Lawrence Byrd and Chris Mellish, Special Purpose, but Domain Independent, Inference Mechanisms, *Proc. ECAI-82*, Orsay, France, 1982, 67-74.

[Cha78] Dennis dw Champeaux, A Theorem Prover Dating a Semantic Network, *Proc. AISB/GI*, Hamburg, West Germany, 1978, 82-92.

[EHL80] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser and D. Raj Reddy, The HEARSAY-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty, *Computing Surveys 12*, 2 (1980), 213-253.

[HaS86] Johannes de Haan and Lenhart K. Schubert, Inference in a Topically Organized Semantic Net, *Proc. AAAI-86 I*, (1986), 334-338.

[Haa86] Johannes de Haan, Inference in a Topically Organized Semantic Net, M.Sc. Thesis, Department of Computing Science, University of Alberta, 1986.

[HWL83] F. Hayes-Roth, D.A. Waterman and D.B. Lenat, in *Building Expert Systems*, Addison-Wesley Publishing Company, Inc., U.S.A., 1983.

[KlB85] Johan de Kleer and John Seely Brown, A Qualitative Physics Based on Confluences, in *Formal Theories of the Commonsense World*, J.R. Hobbs and R.C. Moore (ed.), Ablex, Norwood, NJ, 1985, 109-183.

[Lad86] Peter Ladkin, Time Representation: A Taxonomy of Interval Relations, *Proc. AAAI-86 I*, (1986), 360-366.

[LMF86] Bruce Leban, David D. McDonald and David R. Forster, A Representation for Collections of Temporal Intervals, *Proc. AAAI-86 I*, (1986), 367-371.

[NeO79] Greg Nelson and Derek C. Oppen, Simplification by Cooperating Decision Procedures, *ACM Transactions on Programming Languages and Systems 1*, 2 (1979), 245-257.

[Nii86] H. Penny Nii, Blackboard Application Systems, Blackboards Systems from a Knowledge Engineering Perspective, *The AI Magazine*, 1986, 82-106. August issue.

[Pap82] Mary-Angela Papalaskaris, Special Purpose Inference Methods, M.Sc. Thesis, Department of Computing Science, University of Alberta, 1982.

[PaS82] M.A. Papalaskaris and L.K. Schubert, " Inference, Incompatible Predicates and Colors", *Proc. Fourth Biennial Conf. CSCSI/SCIEO*, 1982, 97-102.

[Pat85] Peter F. Patel-Schneider, A Decidable First-Order Logic for Knowledge Representation, *Proceedings of IJCAI-85 I*, (1985), 455-458.

[Ric82] Charles Rich, Knowledge Representation Languages and Predicate Calculus: How to Have Your Cake and Eat It Too, *Proc. AAAI-82*, Pittsburgh, Penn., 1982, 193-196.

[Ric85] C. Rich, The Layered Architecture of a System for Reasoning about Programs, *Proceedings of IJCAI-85 I*, (1985), 540-546.

[SPT83] Lenhart K. Schubert, Mary Angela Papalaskaris and Jay Taugher, Determining Type, Part, Color, and Time Relationships, *IEEE Computer 16*, 10 (1983), 53-60.

[SPT87] L.K. Schubert, M.A. Papalaskaris and J. Taugher, Accelerating Deductive Inference: Special Methods for Taxonomies, Colours and Times, in *The Knowledge Frontier*, N. Cercone and G. McCalla (ed.), 1987.

[Sho84] Robert E. Shostak, Deciding Combinations of Theories, *Journal of the ACM 31*, 1 (1984), 1-12.

[Sti83] Mark E. Stickel, Theory Resolution: Building in Nonequational Theories, *Proc. AAAI-83*, Washington, D.C., 1983, 391-397.

[Sti85] Mark E. Stickel, Automated Deduction by Theory Resolution, *Proceedings of IJCAI-85 2*, (1985), 1181-1186.

[Tau83] J. Taugher, An Efficient Representation for Time Information, M.Sc. Thesis, Department of Computing Science, University of Alberta, 1983.

[Vil82] Marc B. Vilain, A System for Reasoning About Time, *Proc. AAAI-82*, Pittsburgh, Penn., 1982, 197-201.

[Vil85] Marc Vilain, The Restricted Language Architecture of a Hybrid Representation System, *Proceedings of IJCAI-85 I*, (1985), 547-551.

[ViK86] Marc Vilain and Henry Kautz, Constraint Propagation Algorithms for Temporal Reasoning, *Proc. AAAI-86 I*, (1986), 377-382.

[Wal83] Christoph Walther, A Many-Sorted Calculus Based on Resolution and Paramodulation, *Proc. IJCAI-83*, Karlsruhe, West Germany, 1983, 882-891.

Appendix 1

Analysis of the Number of Cross Chain Links

Each point on each chain can have a cross link to each point on every other chain. If n_i is the number of nodes on chain i, c is the total number of chains, and N is the total number of nodes in the graph:

$$
\begin{aligned}
m &= \textit{Total number of possible links - in chain links} \\
&= \sum_{i=1}^{c} n_i * (\sum_{j=1}^{c} n_j) - \sum_{i=1}^{c} n_i * n_i \\
\text{but } \sum_{i=1}^{c} n_i * (\sum_{j=1}^{c} n_j) &= \sum_{i=1}^{c} n_i * N = N * \sum_{i=1}^{c} n_i = N^2 \\
\text{so } m &= N^2 - \sum_{i=1}^{c} n_i^2 \quad < N^2
\end{aligned}
$$

• The larger the chains (n), the more difference there will be between m and N^2.

Now considering the case where the chains are of equal length to learn about the role of c:

$$
\begin{aligned}
m &= N^2 - \sum_{i=1}^{c} n_i^2 \\
&= N^2 - \sum_{i=1}^{c} n^2 \quad (n \text{ is a constant here}) \\
&= (c*n)^2 - n^2 \sum_{i=1}^{c} 1 \\
&= c^2 * n^2 - c * n^2
\end{aligned}
$$

Alternatively, from each point, there are $(c-1)*n$ possible cross chain links it can have. Since there are $c*n$ points, there are $n^2*c*(c-1)$ possible cross chain links.

$$
\begin{aligned}
m &< N^2 \\
n^2*c*(c-1) &< (c*n)^2 \\
n^2*c*(c-1) &< c^2*n^2 \\
c*(c-1) &< c^2 \\
c^2-c &< c^2
\end{aligned}
$$

• The smaller the number of chains (c), the greater the relative difference between N^2 and m.

So the smallest values of m are obtained with a few long chains. In the expected case, one of the chains contains 70% of the nodes (experimentally determined by Taugher [Tau83]), so (letting chain number c be this chain):

$$
\begin{aligned}
m &= N^2 - \sum_{i=1}^{c} n_i^2 \\
&= N^2 - (.7N)^2 - \sum_{i=1}^{c-1} n_i^2 \\
&<= N^2 - .49N^2 \\
&<= .51N^2
\end{aligned}
$$

Appendix 2
Closure Algorithm for Taugher and Schubert Model

One of the uses of computing transitive closure is that any inconsistencies will be detected. In [SPT87], Schubert et al suggest that consistency determination, done incrementally as each relation is entered, could be done is $O(mr)$ time, where m is the number of cross chain links, and r is the number of relations, (each of the r relations takes $O(m)$ time to check its consistency). Alternatively, one could compute the transitive closure itself[19].

```
Initialize
        For each relation given
          Enter according to Taugher entry methods (O(1) in expected case)
          Table[i,j] = Table[i,j] +[20] R<i,j>
          If (chain(i) <> chain(j))
            If table entry changed
              Place <i,j> on Fifo Queue
            Table[j,i] = Table[j,i] + InverseR<i,j>
            If it changes
              Place <j,i> on Fifo Queue

Propagate within chain
        For every chain c from 1 to c
          For every i in chain c
            For every j in chain c
              If i is before j then
                Table[i,j] = Table[i,j] + (<=)
                Table[j,i] = Table[j,i] + (>=)
              If i is after j then
                Table[i,j] = Table[i,j] + (>=)
                Table[j,i] = Table[j,i] + (<=)

Propagate(i,j)
        For every k in the set of points
          If chain(k) ~= chain(i)
            Table[i,k] = Table[i,k] + (Table[i,j] x Table[j,k])
            If it changes
              Place on Fifo Queue
          If chain(k) ~= chain(j)
            Table[k,j] = Table[k,j] + (Table[k,i] x Table[i,j])
            If it changes
              Place on Fifo Queue
```

[19] This algorithm is based on the closure algorithm for time point algebra given in [ViK86], with changes to exploit the Taugher and Schubert representation. The original algorithm does not include inverse relations, nor is there any way to calculate them during the closure calculations. This algorithm has been modified to incorporate those inverses for use immediately.

[20] The addition relation here is the point relation addition described by Vilain and Kautz. It calculates the resulting relation vector between two points using the relation between the first point an an intermediate, and the intermediate and the second point. If this calculation results in the null vector, the set of relations are inconsistent.

Analysis

Propagate within chain - $O(c*n^2)$

For each item on queue - $O(N)$

Number of queue items - Maximum of m (number of cross chain links)
- only ones that go on the queue, and can be constrained only a finite # times.

$\Rightarrow O(m*N+c*n^2)$

Note that this is still $O(n^3)$, but in the expected case it can do significantly better than the asymptotic.

Some example timings for different values of c and n

Maximum Timings for $N=12, N^3=1728$					
c	n	m	$c*n^2$	$m*N$	$c*n^2+m*N$
1	12	-	144	0	144
2	6	72	72	864	936
3	4	96	48	1152	1200
4	3	108	36	1296	1332
6	2	120	24	1440	1464
12	1	144	-	1728	1728

Appendix 3

Example Concept Hierarchy

```
add hier tp.spec physical-object abstract-object

add hier physical-object living-thing non-living-thing
add hier   living-thing plant creature
add hier     plant leafy-plant fungus cactus
add hier       leafy-plant tough-stemmed-plant soft-stemmed-plant
add hier         tough-stemmed-plant tree shrub
add hier         soft-stemmed-plant alfalfa orchid yucca grain

add hier     creature person animal
add hier       person adult minor
add hier         adult man woman
add hier         minor adolescent child infant
add hier           child girl boy

add hier       animal microbe bug larger-animal
add hier         bug caterpillar snail
add hier         larger-animal fishlike-animal birdlike-animal reptile
add hier         larger-animal warm-blooded-quadraped simian
add hier           birdlike-animal bird
add hier           warm-blooded-quadraped wolf fox

add hier  non-living-thing inanimate-natural-object artifact food
add hier    inanimate-natural-object astronomical-object rock mountain
add hier    inanimate-natural-object lake cloud forest natural-shelter
add hier    inanimate-natural-object natural-materiel
add hier         natural-shelter den cave
add hier      natural-materiel wood straw wool

add hier    artifact device furniture container conduit musical-instrument
add hier    artifact art-object money document architectural-object
add hier    artifact covering-object

add hier      device vehicle hand-tool heavy-machinery weapon
add hier      device communication-device computer sensing-measuring-device
add hier      device writing-device fastening-device recreational-device
add hier      device lamp household-appliance eating-device grooming-device
add hier        hand-tool axe
add hier        eating-device butter-knife fork spoon

add hier      furniture bed table chair mattress

add hier      container basket

add hier      architectural-object building barrier tower bridge
add hier      architectural-object tunnel paved-area
```

```
add hier          building cottage

add hier        covering-object apparel tablecloth cloth
add hier          apparel nightgown nightcap cape hood

add hier      food person-food
add hier        person-food baked-food natural-food meat
add hier          baked-food pie cake cookie tart
add hier          natural-food apple honey

add hier abstract-object Integer12 Integer18

add hier tp.topics tp.part tp.physical-quality tp.abstract-quality
add hier tp.topics tp.static-relationship tp.function tp.behavior
add hier tp.topics tp.episode tp.type

add hier    tp.part tp.external-part tp.internal-part

add hier    tp.physical-quality tp.size tp.external-quality tp.mass
add hier    tp.physical-quality tp.substance

add hier      tp.external-quality tp.appearance tp.odour tp.tactile-quality

add hier        tp.appearance tp.form tp.coloring tp.translucency tp.covering

add hier          tp.coloring yellow green blue purple red orange black brown
add hier          tp.coloring grey white pink

add hier          tp.covering tp.apparel

add hier        tp.tactile-quality tp.texture tp.hardness tp.resilience

add hier    tp.abstract-quality tp.mental-quality tp.existance tp.name
add hier      tp.mental-quality tp.emotional-disposition
add hier      tp.mental-quality tp.intellectual-disposition

add hier    tp.static-relationship tp.physical-relationship
add hier    tp.static-relationship tp.emotional-relationship
add hier    tp.static-relationship tp.abstract-relationship

add hier      tp.physical-relationship tp.location tp.part-of tp.containment
add hier      tp.physical-relationship tp.force tp.affinity tp.size-relationship

add hier        tp.part-of tp.is-part-of tp.has-part
add hier        tp.size-relationship tp.size-to tp.size-from

add hier    tp.emotional-relationship tp.emotional-giver
add hier      tp.emotional-relationship tp.emotional-receipient

add hier      tp.abstract-relationship tp.kinship tp.control tp.ownership
```

```
add hier    tp.abstract-relationship tp.membership tp.arithmetic-relationship
add hier    tp.ownership tp.owner tp.owned-by

add hier  tp.function tp.passive-function tp.active-function

add hier  tp.behavior tp.locomation tp.self-maintenance tp.constructive-activity
add hier  tp.behavior tp.social tp.development

add hier    tp.self-maintenance tp.feeding tp.elimination
add hier    tp.self-maintenance tp.cleaning-and-grooming tp.sleep tp.shelter
add hier      tp.feeding tp.feeding-likes

add hier    tp.social tp.communication tp.ritual tp.mating tp.fighting tp.play
```

Appendix 4

Example Topic Hierarchy

```
add topic tp.gen physical-object
add topic tp.gen   living-thing
add topic tp.gen     plant fungus cactus
add topic tp.gen       leafy-plant
add topic tp.gen         tough-stemmed-plant tree shrub
add topic tp.gen         soft-stemmed-plant alfalfa orchid yucca grain

add topic tp.gen     creature
add topic tp.gen       person
add topic tp.gen         adult man woman
add topic tp.gen         minor adolescent infant
add topic tp.gen           child girl boy

add topic tp.gen       animal microbe
add topic tp.gen         bug caterpillar snail
add topic tp.gen         larger-animal fishlike-animal reptile
add topic tp.gen           birdlike-animal bird
add topic tp.gen           warm-blooded-quadraped wolf fox
add topic tp.gen           simian

add topic tp.gen   non-living-thing
add topic tp.gen     inanimate-natural-object astronomical-object rock mountain
add topic tp.gen       lake cloud forest
add topic tp.gen       natural-shelter cave den
add topic tp.gen       natural-materiel wood straw wool

add topic tp.gen     artifact conduit musical-instrument
add topic tp.gen       art-object money document

add topic tp.gen       device vehicle heavy-machinery weapon
add topic tp.gen         communication-device computer sensing-measuring-device
add topic tp.gen         writing-device fastening-device recreational-device
add topic tp.gen         lamp household-appliance grooming-device
add topic tp.gen         hand-tool axe
add topic tp.gen         eating-device butter-knife fork spoon

add topic tp.gen       furniture bed chair table mattress

add topic tp.gen       container basket

add topic tp.gen       architectural-object barrier tower bridge tunnel paved-area
add topic tp.gen         building cottage

add topic tp.gen       covering-object tablecloth cloth
add topic tp.gen         apparel nightgown nightcap cape hood
```

```
add topic tp.gen     food
add topic tp.gen       person-food meat
add topic tp.gen         baked-food pie cake cookie tart
add topic tp.gen         natural-food apple honey

add topic tp.part null

add topic tp.physical-quality age-of 2

add topic    tp.size small big large huge

add topic    tp.external-quality null

add topic      tp.appearance null
add topic        tp.form sharp long thin pointy fat

add topic      tp.coloring yellow green blue purple red orange black brown
add topic      tp.coloring grey white pink pale-skinned

add topic      tp.translucency clear hazy opaque

add topic      tp.covering cover 2
add topic        tp.apparel wear 1

add topic      tp.tactile-quality null
add topic        tp.texture hairy smooth-skinned woven-with 1

add topic    tp.substance made-of 1

add topic tp.abstract-quality nul
add topic    tp.mental-quality animate
add topic      tp.emotional-disposition fierce
add topic      tp.intellectual-disposition clever
add topic    tp.existance kill 2 make 2
add topic    tp.name name-of 2

add topic tp.static-relationship null

add topic    tp.physical-relationship joined-to attached-to
add topic      tp.location in 1 on 1 wear 2 cover 1 transport 2
add topic      tp.part-of null
add topic        tp.is-part-of leg-of 1 foot-of 1 body-of 1 head-of 1 eye-of 1
add topic        tp.is-part-of neck-of 1 toe-of 1 nose-of 1 breast-of 1
add topic        tp.is-part-of tail-of 1 teeth-of 1 mouth-of 1 finger-of 1
add topic        tp.is-part-of arm-of 1 ear-of 1 part-of 1
add topic        tp.has-part leg-of 2 foot-of 2 body-of 2 head-of 2 eye-of 2
add topic        tp.has-part neck-of 2 toe-of 2 nose-of 2 breast-of 2
add topic        tp.has-part tail-of 2 teeth-of 2 mouth-of 2 finger-of 2
add topic        tp.has-part arm-of 2 ear-of 2 part-of 2
add topic      tp.containment in 2 live-in 2
```

```
add topic     tp.force null
add topic     tp.affinity null
add topic     tp.size-relationship null
add topic       tp.size-to smaller-than 1
add topic       tp.size-from smaller-than 2

add topic   tp.emotional-relationship null
add topic     tp.emotional-giver     likes 1 loves 1 hates 1
add topic     tp.emotional-receipient likes 2 loves 2 hates 2

add topic   tp.abstract-relationship null
add topic     tp.kinship grandmother-of mother-of sister-of father-of
add topic     tp.kinship parent-of grandfather-of brother-of grandparent-of
add topic     tp.control null
add topic     tp.ownership null
add topic       tp.owner belongs-to 2
add topic       tp.owned-by belongs-to 1
add topic     tp.membership member-of member-of-0 equal
add topic     tp.arithmetic-relationship less-than greater-than lt gt
add topic     tp.arithmetic-relationship lt= gt= equal

add topic tp.function null
add topic   tp.passive-function sleep-in 2
add topic   tp.active-function transport 1 use 2

add topic tp.behavior use 1 make 1

add topic   tp.self-maintenance kill 1
add topic     tp.feeding eat 1 drink 1
add topic       tp.feeding-likes like-to-eat 1
add topic     tp.elimination null
add topic   tp.self-maintenance null
add topic     tp.cleaning-and-grooming null
add topic     tp.sleep null
add topic     tp.shelter live-in 1

add topic   tp.social null
add topic     tp.communication null
add topic     tp.ritual null
add topic     tp.mating null
add topic     tp.fighting null
add topic     tp.play null

add topic tp.episode meet

add topic tp.type Caucasian male female hunter woodcutter

add topic tp.episode equal before before-0 before-1 after after-0 after-1
add topic tp.episode during during-0 during-1 during--0 during--1
add topic tp.episode during-0-0 during-0-1 during-1-0 during-1-1
```

```
add topic tp.episode contains contains-0 contains-1 contains--0 contains--1
add topic tp.episode contains-0-0 contains-0-1 contains-1-0 contains-1-1
add topic tp.episode overlaps overlaps-0 overlaps-1 overlaps--0 overlaps--1
add topic tp.episode overlaps-0-0 overlaps-0-1 overlaps-1-0 overlaps-1-1
add topic tp.episode overlapped-by overlapped-by-0 overlapped-by-1 overlapped-by--0 overlapped-by--1
add topic tp.episode overlapped-by-0-0 overlapped-by-0-1 overlapped-by-1-0 overlapped-by-1-1
add topic tp.episode between between-0 between-1 between--0 between--1
add topic tp.episode between-0-0 between-0-1 between-1-0 between-1-1
add topic tp.episode at-most-before at-least-before exactly-before
add topic tp.episode at-most-after at-least-after exactly-after
```

Appendix 5

General Knowledge Used in Examples

```
set assert major-imp
(A x A y A z_ep (x eat y z)      => (x living-thing))
(A x A y A z_ep (x in y z)       => (x smaller-than y))
(A x A y A z_ep (x live-in y z)  => (x smaller-than y))
(A x A y A z_ep (x live-in y z)  => (x living-thing))
(A x A y A z_ep (x live-in y z)  => (x in y))
(A x A y A z_ep (x kill y z)     => (y living-thing))
(A x A y A z_ep (x cover y z)    => (x on y))
(A x A y A z_ep (x meet y z)     => (y meet x z))

(A x A y A z_ep A w_ep ((x kill y z) & (w before z)) => (y alive w))
(A x A y A z_ep A w_ep ((x kill y z) & (w after z)) => (~ y alive w))
set assert nomajor

(A x (x fat) => (~ x thin))

(A x E y E z
  ((x person) | (x larger-animal)) =>
          ((y head-of x) & (z body-of x)))
(A x (x creature) => (x animate))

(A x E y (x person) => (y teeth-of x))
(A x A y ((x person) & (y teeth-of x)) => ((~ y sharp) & (~ y big) &
                                  (~ y pointy) & (y white)))

(A x E y ((x person) => (y eye-of x)))
(A x A y (((x person) & (y eye-of x)) => ((~ y big))))


(A x E y (x person) => (y ear-of x))
(A x A y ((x person) & (y ear-of x)) => ((~ y big) & (~ y hairy)))

(A x (x person) => (A y (y person-food) => (x like-to-eat y)))
(A x (x person) => (A y (~ y person-food) => (~ x like-to-eat y)))

(A x (x person) => (x smooth-skinned))
(A x ((x person) & (x Caucasian)) => (x pale-skinned))

(A x (x person) => (E y (y father-of x) & (y man)))
(A x (x person) => (E y (y mother-of x) & (y woman)))

(A x A y (x person) => (E x (x building) & (x live-in y)))

(A x A y ((x child) & (y age-of x)) => (y less-than Integer12))
(A x A y A z A w_ep ((x child) & ((y father-of x) | (y mother-of x)) &
        (z building) & (y live-in z w)) => (x live-in z w))
```

```
(A x A y ((x minor) & (y age-of x)) => (y less-than Integer18))
(A x A y ((x adult) & (y age-of x)) => (y greater-than Integer18))

(A x (x girl) => (x female))
(A x (x boy) => (x male))
(A x (x woman) => (x female))
(A x (x man) => (x male))

(A x (x wolf) => (x grey))

(A x E y (x wolf) => (y ear-of x))
(A x A y ((x wolf) & (y ear-of x)) => ((y big) & (y hairy)))

(A x E y (x wolf) => (y teeth-of x))
(A x A y ((x wolf) & (y teeth-of x)) => ((y sharp) & (y big) &
                        (y pointy) & (y white)))

(A x E y (x wolf) => (y eye-of x))
(A x A y ((x wolf) & (y eye-of x)) => ((y big)))

(A x E y (x wolf) => (y tail-of x))
(A x (x wolf) => (x fierce))
(A x (x wolf) => (x clever))

(A x (x wolf) => (A y (y creature) => (x like-to-eat y)))

(A x (x wolf) => (x hairy))

(A x A y A z_ep (x wolf) => ((x live-in y z) & (y natural-shelter)))

(A x A y ((x basket) & (x made-of y)) => (x woven-with y))
(A x (x basket) => (E y (y straw) & (x made-of y)))
(A x A y A z_ep ((x container) & (y in x z)) => (x transport y z))
(A x A y A z_ep ((x creature) & (y container) & (x transport y z)) => (x transport y z))
(A x A y A z A w_ep ((x transport y w) & (y transport z w)) => (x transport z w))

(A x A y (((x cloth) & (x made-of y)) => (x woven-with y)))
(A x ((x apparel) => (E y (y cloth) & (x made-of y))))
(A x A y A z_ep (((x creature) & (y apparel) & (x wear y z)) => (y cover x z)))

(A x A y A z A w_ep ((x creature) & ((y hood) | (y nightcap)) & (x wear y w) &
      (z head-of x)) => (y cover z w))
(A x A y A z A w_ep ((x creature) & ((y cape) | (y nightgown)) & (x wear y w) &
      (z body-of x)) => (y cover z w))

(A x A y A z_ep ((x creature) & ((y building) | (y natural-shelter)) &
     (x in y z)) => (y shelter x z))

(A x (x device) => (E y E z_ep (y person) & (y use x z)))
```

(A x (x artifact) => (E y E z_ep (y person) & (y make x z)))
(A x (x inanimate-natural-object) => (~ (E y E z_ep ((y person) & (y make x z)))))

(A x (x furniture) => (E y E z_ep (y building) & (x in y z)))

(A x A y A z_ep ((x bed) & (y creature) & (y sleep-in x z)) => (x on y z))

(A x (x baked-food) => (E y E z_ep (y person) & (y make x z)))
(A x (x natural-food) => (E y (y plant) & (x part-of y)))
(A x (x meat) => (E y (y creature) & (x part-of y)))

(A x (x tablecloth) => (E y (y cloth) & (x made-of y)))
(A x (x tablecloth) => (E y E z_ep (y table) & (x cover y z)))

Appendix 6

Little Red Riding Hood Knowledge

(LRRH girl)

(E x E y E z_ep (x cape) & (x red) & (LRRH wear x z) &
(E w (w wool) & (x made-of w)) &
(y hood) & (y red) & (LRRH wear y z) &
(E w (w wool) & (y made-of w)) &
(y attached-to x) & (z equal story_ep))
(GM grandmother-of LRRH)

(GM woman)
(GM live-in COT gm-home_ep)
(gm-home equal story)

(W wolf)

(WOODCUTTER man)

(A x (x person) => (x Caucasian))

(BAS basket)
(E x (x straw) & (BAS made-of x))
(E w E x E y E z E u_ep (w honey) & (w in BAS yummies_ep) &
(x cake) & (x in BAS yummies_ep) &
(y cookie) & (y in BAS yummies_ep) &
(z pie) & (z in BAS yummies_ep) &
(yummies_ep during story))

(E x E y E z E w_ep (x tablecloth) & (x in BAS w) &
(y butter-knife) & (y in BAS w) & (w during story))

(COT cottage)
(GM-BED in COT story)
(E y (y wood) & (COT made-of y))

(GM-BED bed)
(E y (y wood) & (GM-BED made-of y))
(E y (y mattress) & (y on GM-BED))
(E y E x E z_ep (y bedclothes) & (x cloth) & (y made-of x) &
(y on GM-BED z) & (z equal story))

(E x x rock)

(~ LRRH smaller-than BAS) (BAS smaller-than LRRH)
(LRRH smaller-than COT) (~ COT smaller-than LRRH)

```
(LRRH smaller-than GM-BED)      (~ GM-BED smaller-than LRRH)

(~ GM smaller-than BAS)   (BAS smaller-than GM)
(GM smaller-than COT)   (~ COT smaller-than GM)
(GM smaller-than GM-BED)      (~ GM-BED smaller-than GM)

(~ WOODCUTTER smaller-than BAS)    (BAS smaller-than WOODCUTTER)
(WOODCUTTER smaller-than COT)   (~ COT smaller-than WOODCUTTER)
(WOODCUTTER smaller-than GM-BED)      (~ GM-BED smaller-than WOODCUTTER)

(~ W smaller-than BAS)    (BAS smaller-than W)
(W smaller-than COT)   (~ COT smaller-than W)
(W smaller-than GM-BED)      (~ GM-BED smaller-than W)

(BAS smaller-than GM-BED)      (~ GM-BED smaller-than BAS)
(BAS smaller-than COT)   (~ COT smaller-than BAS)
(GM-BED smaller-than COT)      (~ COT smaller-than GM-BED)

(A x (x basket) => (x smaller-than W))
(A x (x basket) => (~ W smaller-than x))

(E x E y E w_event E z_event (x nightgown) & (GM wear x z) & (W wear x w) &
      (y nightcap)  & (GM wear y z) & (W wear y w) )_

; LRRH is carrying a basket through the woods to GM when she meets a wolf

(LRRH transport BAS lrrh-walk_ep)
(LRRH meet W wolf-meet_ep)
(wolf-meet_ep during-1-1 lrrh-walk_ep story)

; The wolf races ahead to the cottage, eats GM, and waits in GM's bed for LRRH

(W in COT wolf-in-cot_ep)
(wolf-in-cot_ep after-1 wolf-meet)
(W eat GM wolf-eat-gm_ep)
(wolf-eat-gm_ep during wolf-in-cot)
(W on GM-BED wolf-in-bed_ep)
(wolf-in-bed_ep after-1 wolf-eat-gm wolf-in-cot)

; When LRRH arrives at the cottages, she finds the wolf in the bed. The wolf
; eats her

(LRRH in COT lrrh-in-cot_ep)
(lrrh-in-cot_ep after-0 lrrh-walk)
((start-of lrrh-in-cot) during wolf-in-bed)
(W eat LRRH wolf-eat-lrrh_ep)
(wolf-eat-lrrh_ep after-0 wolf-in-bed wolf-in-cot)
(wolf-eat-lrrh during-1 lrrh-in-cot)

; A woodcutter hears the screams and races into the cottage and kills the wolf
```

```
(WOODCUTTER in COT wc-in-cot_ep)
(WOODCUTTER kill W wolf-demise_ep)
(wc-in-cot_ep after wolf-eat-lrrh)
(wolf-demise_ep during-1-0 wolf-in-cot_ep)
(wolf-demise during wc-in-cot)

; GM and WC and LRRH all eat the goodies in the basket until LRRH and WC leave

(yummies before-0 eat-goodies_ep)
(A x A z_ep ((x person-food) & (x in BAS yummies) )
                        => ( (GM eat x eat-goodies_ep) &
                           (LRRH eat x eat-goodies_ep) &
                           (WOODCUTTER eat x eat-goodies_ep) ))

(eat-goodies during--0 lrrh-in-cot story)
(eat-goodies during--0 wc-in-cot)
(eat-goodies_ep after-1 wolf-demise)

; They talked while eating goodies

(LRRH talk all-talk_ep)
(GM talk all-talk_ep)
(WOODCUTTER talk all-talk_ep)
(all-talk during eat-goodies)
```

Appendix 7

Sample Run of ECoNet

Note: user input is in bold print; comments in italics; system output in regular print.

```
Specialist HIER-SPECIALIST is now active
#P"/usr/sunkay/grad/steph/lisp/net.lisp"
==> set noecho
==> set assert nodisplay-id
==> <eg.hier
==> <eg.indicate
==> use-spec color time number set
Specialist COLOR-SPECIALIST is now active
Specialist TIME-SPECIALIST is now active
Specialist NUMBER-SPECIALIST is now active
Specialist SET-SPECIALIST is now active
(COLOR-SPECIALIST TIME-SPECIALIST NUMBER-SPECIALIST SET-SPECIALIST)
==> tweak spec-effort entry 0
Setting specialist entry effort to 0
==> tweak spec-effort eval 0
Setting specialist evaluation effort to 0
==> <eg.general
==> <eg.lrrh
==> set echo on
==> set assert display-id
==> tweak spec-effort eval
Setting specialist evaluation effort to 7
==> tweak spec-effort entry
Setting specialist entry effort to 7

==> tweak qa-iterations 100
==> tweak qa-access-weight 30

==> tweak qa-flag (false-true off) (true-true off)
==> trace qa-time
(QA-TIME)
==> display qa-flag
QA Evaluation Flags:
   True       ON      - infer literal 1
   False      ON      - infer rest of clause 1
   True-True  OFF     - infer literal 2
   True-False ON      - infer rest of clause 2
   False-True OFF     - infer literal2
   False-False ON     - infer rest of clause 2

==> ?(E x E z_ep (x cake) & (LRRH eat x z))
   Used .76 seconds CPU time  .96 seconds real time
YES
```

```
==> ?(E x E y E z_ep (x creature) & (y creature) & (x eat y z))
  Used .9 seconds CPU time   1.06 seconds real time
YES

==> ?(E x E y E z_ep (x person) & (y wolf) & (x kill y z))
  Used 14.18 seconds CPU time 15.4 seconds real time
YES

==> ?(E x E y E z_ep (x person) & (y wolf) & (x kill y z))
  Used .38 seconds CPU time   .38 seconds real time
YES

==> ?(E x E z_ep (x rock) & (x eat GM z))
  Used .22 seconds CPU time   .22 seconds real time
NO

==> ?(E x E z_ep (x basket) & (W live-in x z))
  Used .4 seconds CPU time   .38 seconds real time
NO

==> ?(E x E z_ep (x red) & (LRRH wear x z))
  Used .26 seconds CPU time   .26 seconds real time
YES

==> ?(A x (x wolf) => (x grey))
  Used .04 seconds CPU time   .04 seconds real time
YES

==> ?(A x (x cape) => (x blue))
  Used .14 seconds CPU time   .14 seconds real time
NO

==> ?(E x E z_ep W eat x z)
  Used .14 seconds CPU time   .14 seconds real time
YES

==> ?(A x (x rock) => (x person))
  Used .3 seconds CPU time   .3 seconds real time
NO
```

```
==> ?(E x E y E z_ep (x creature) & (x kill y z) & (y grey))
  Used .42 seconds CPU time  .4 seconds real time
YES

==> ?(E z_ep BAS eat GM z)
  Used .16 seconds CPU time  .16 seconds real time
NO

==> ?(E x E y E z_ep (x kill y z) & (y creature))
  Used .36 seconds CPU time  .36 seconds real time
YES

==> ?(E x E y E z_ep (x kill y z) & (y grey))
  Used .32 seconds CPU time  .32 seconds real time
YES

==> ?(E z_ep W kill BAS z)
  Used .66 seconds CPU time  .66 seconds real time
NO

==> ?(E x x grey)
  Used .18 seconds CPU time  .18 seconds real time
YES

==> ?(E x x red)
  Used .16 seconds CPU time  .18 seconds real time
YES

==> ?(LRRH rock)
  Used .06 seconds CPU time  .08 seconds real time
NO

==> ?(E z_ep BAS live-in COT z)
  Used .36 seconds CPU time  .48 seconds real tim
NO

==> ?(E x E z_ep LRRH wear x z)
  Used .48 seconds CPU time  .56 seconds real time
YES
```

```
==> ?(E x E z_ep (x hood) & (LRRH wear x z))
   Used .32 seconds CPU time   .36 seconds real time
YES


==> ?(W like-to-eat LRRH)
   Used .12 seconds CPU time   .14 seconds real time
YES


==> ?(A x (x cake) => (LRRH like-to-eat x))
   Used .38 seconds CPU time   .48 seconds real time
YES


==> ?(E x E z_ep (x in BAS z) & (LRRH like-to-eat x))
   Used 1.08 seconds CPU time  1.08 seconds real time
YES


==> ?(E x E z_ep (x in BAS z) & (~ LRRH like-to-eat x))
   Used 1.44 seconds CPU time  1.44 seconds real time
YES


==> ?(A x A z_ep (x in BAS z) => (LRRH like-to-eat x))
   Used 16.56 seconds CPU time 20.02 seconds real time
NO


==> ?(LRRH like-to-eat BAS)
   Used .24 seconds CPU time   .24 seconds real time
NO

; did the wolf get killed while in the cottage
==> ?(wolf-demise during wolf-in-cot)
   Used 1.64 seconds CPU time  1.88 seconds real time
YES
==> ?(E y E z_ep (y kill W z) & (z during wolf-in-cot))
   Used 290.16 seconds CPU time       316.024 seconds real time
YES

==> ?(wolf-in-cot contains wolf-demise)
   Used 2.78 seconds CPU time  2.78 seconds real time
YES
==> ?(E x E y_ep E z_ep (x kill W y) & (W in COT z) & (z contains y))
   Used 191.74 seconds CPU time       197.08 seconds real time
YES
```

```
; did the wolf eat gm after eating lrrh?
==> ?(wolf-eat-gm after-1 wolf-eat-lrrh)
   Used 2.3 seconds CPU time  6.34 seconds real time
NO
==> ?(A x_ep A y_ep ((W eat LRRH x) & (W eat GM y)) => (y after-1 x))
   Used 74.48 seconds CPU time 79.84 seconds real time
NO

; did the wolf eat lrrh after eating gm?
==> ?(wolf-eat-lrrh after-1 wolf-eat-gm)
   Used 11.9 seconds CPU time  12.38 seconds real time
YES
==> ?(E x_ep E y_ep (W eat LRRH x) &( W eat GM y) & (x after-1 y))
   Used 77.18 seconds CPU time 82.64 seconds real time
YES


==> trace time-eval
(TIME-EVALUATION)


==> ?((relation wolf-in-cot lrrh-in-cot) is what)
   Time Specialist: Found relation WOLF-IN-COT OVERLAPS-1-1 LRRH-IN-COT
   Used 5.64 seconds CPU time  5.64 seconds real time
UNKNOWN
==> ?(E x_ep E y_ep (W in COT x) & (LRRH in COT y) & ((relation x y) is what))
   Time Specialist: Found relation WOLF-IN-COT OVERLAPS-1-1 LRRH-IN-COT
   Used 7.28 seconds CPU time  7.6 seconds real tim
UNKNOWN


==> ?((relation lrrh-in-cot wc-in-cot) is what)
   Time Specialist: Found relation LRRH-IN-COT CONTAINS-1-0 WC-IN-COT
   Used 4.46 seconds CPU time  4.46 seconds real time
UNKNOWN
==> ?(E x_ep E y_ep (WOODCUTTER in COT x) & (LRRH in COT y) & ((relation y x) is what))
   Time Specialist: Found relation LRRH-IN-COT CONTAINS-1-0 WC-IN-COT
   Used 20.48 seconds CPU time 21.12 seconds real time
UNKNOWN


==> untrace time-eval
(TIME-EVALUATION)

; was the wolf alive when everyone was eating goodies?
?(E x_ep (W alive x) & (x during eat-goodies))
   Used 826.3 seconds CPU time 843.344 seconds real time
?
   Used 924.26 seconds CPU time      941.544 seconds real time
NO
```

```
; was the wolf alive when he ate gm?
?(E x_ep E y_ep (W eat GM x) & (W alive y) & (y equal x) )
   Used 114.12 seconds CPU time      116.224 seconds real time
YES

==> trace time-fac
(TIME-FACTORABLE)
;did anything that happened while the wolf was in bed happen while
; he was in the cottage?
?(A x_ep (x during wolf-in-bed) => (x during wolf-in-cot))
   Time Specialist: Trying to factor (EPISODE-VAR-1 DURING WOLF-IN-COT)
            and (~ EPISODE-VAR-1 DURING WOLF-IN-BED)
   Time Specialist: Factored to (WOLF-IN-BED DURING WOLF-IN-COT)
   Used 68.32 seconds CPU time 68.56 seconds real time
YES
==> untrace time-fac
(TIME-FACTORABLE)


;could/did anything happen between lrrh's walk and her being in the cottage?
?(E x_ep (x between-1-1 lrrh-walk lrrh-in-cot))
   Used .48 seconds CPU time  .5 seconds real time
NO

; did everything happen while the wolf was in bed?
?(A x_ep (x during wolf-in-bed))
   Used 195.0 seconds CPU time 202.156 seconds real time
NO

;did the woodcutter kill the wolf before they all ate the goodies in the basket?
?(E x_ep (WOODCUTTER kill W x) & (x before eat-goodies))
   Used 367.68 seconds CPU time      376.918 seconds real time
YES

;did the woodcutter kill the wolf before the wolf got into the cottage?
==> trace unify
(UNIFICATION)
==> ?(E x_ep (WOODCUTTER kill W x))
   Substituting WOLF-DEMISE/EPISODE-VAR-1 in (WOODCUTTER W EPISODE-VAR-1) and
   nothing in (WOODCUTTER W WOLF-DEMISE) will unify the two.

   Used .34 seconds CPU time  .34 seconds real time
YES
==> untrace unify
(UNIFICATION)
==> ?(E x_ep (WOODCUTTER kill W x) & (~ x equal wolf-demise))
   Used 316.44 seconds CPU time      330.11 seconds real time
==> trace unify
(UNIFICATION)
==> ?(E x_ep (W in COT x))
```

```
  Substituting WOLF-IN-COT/EPISODE-VAR-1 in (W COT EPISODE-VAR-1) and
  nothing in (W COT WOLF-IN-COT) will unify the two.

  Used .44 seconds CPU time  .52 seconds real time
YES
==> untrace unify
(UNIFICATION)
==> ?(E x_ep (W in COT x) & (~ x equal wolf-in-cot))
  Used 276.04 seconds CPU time      287.734 seconds real time
UNKNOWN
==> ?(wolf-demise before wolf-in-cot)
  Used .96 seconds CPU time  .96 seconds real time
NO

==> trace duration-eval search abs-time-eval
(TIME-DURATION-EVALUATION TIME-SEARCH ABSOLUTE-TIME-EVALUATION)

==> (t1_time at-most-before e1_ep '3600)   ; duration of 1 hour
[930]
==> (t1 exactly-after t2_time '7200)
[932]
==> (t2 at-least-after t3_time '1800)
[934]
==> ((start-of e1) before (date '1987 '6 '1 '12 '30 '0))
  Time Specialist: Found relation E1START UNKNOWN (TIME 1987 6 1 12 30 0) using absolute times
  Time Specialist: Found relation E1START UNKNOWN (TIME 1987 6 1 12 30 0) using absolute times
[937]
==> ((end-of e1) after (date '1987 '6 '1 '13 '0 '0))
  Time Specialist: Found relation E1END UNKNOWN (TIME 1987 6 1 13 0 0) using absolute times
  Time Specialist: Found relation E1END UNKNOWN (TIME 1987 6 1 13 0 0) using absolute times
[942]

; now for some evaluations

==> trace time-res time-fac unify
(TIME-RESOLVABLE TIME-FACTORABLE UNIFICATION)
==> ?((duration t3 (end-of e1)) is what)
  Time Specialist: Searching for duration from T3 to E1END
  Time Specialist: Using link between T3 and T2
    with duration (1800 +INFINITY)
  Time Specialist: Searching for duration from T2 to E1END
  Time Specialist: Using link between T2 and T1
    with duration (7200 7200)
  Time Specialist: Searching for duration from T1 to E1END
  Time Specialist: Using link between T1 and E1START
    with duration (-INFINITY 3600)
  Time Specialist: Searching for duration from E1START to E1END
  Time Specialist: Using link between E1START and E1END
    with duration (1800 +INFINITY)
  Time Specialist: Found duration between T3 and E1END 10800 +INFINITY
```

```
  Used 14.34 seconds CPU time 14.96 seconds real time
UNKNOWN

==> ?(t3 at-least-before t1 '8000)
  Time Specialist: Searching for duration from T3 to T1
  Time Specialist: Using link between T3 and T2
    with duration (1800 +INFINITY)
  Time Specialist: Searching for duration from T2 to T1
  Time Specialist: Using link between T2 and T1
    with duration (7200 7200)
  Time Specialist: Found duration between T3 and T1 9000 +INFINITY
  Used .16 seconds CPU time  .16 seconds real time
YES
==> ?(t1 at-least-before t3 '11000)
  Time Specialist: Searching for duration from T1 to T3
  Time Specialist: Using link between T1 and E1START
    with duration (-INFINITY 3600)
  Time Specialist: Searching for duration from E1START to T3
  Time Specialist: Using link between E1START and E1END
    with duration (1800 +INFINITY)
  Time Specialist: Searching for duration from E1END to T3
  Time Specialist: Found duration between T1 and T3 -INFINITY +INFINITY
  Used .48 seconds CPU time  .46 seconds real time
NO
==> ?(e1 at-most-after t2 '110000)
  Time Specialist: Searching for duration from T2 to E1START
  Time Specialist: Using link between T2 and T1
    with duration (7200 7200)
  Time Specialist: Searching for duration from T1 to E1START
  Time Specialist: Using link between T1 and E1START
    with duration (-INFINITY 3600)
  Time Specialist: Found duration between T2 and E1START 7200 10800
  Used .14 seconds CPU time  .14 seconds real time
YES
==> ?(t2 at-most-after e1 '8000)
  Time Specialist: Searching for duration from E1END to T2
  Time Specialist: Found duration between E1END and T2 -INFINITY -9000
  Used .12 seconds CPU time  .12 seconds real time
YES

==> untrace
(UNIFICATION TIME-FACTORABLE TIME-RESOLVABLE ABSOLUTE-TIME-EVALUATION
 TIME-SEARCH TIME-DURATION-EVALUATION QA-TIME)

;;; Subnet test

==> (John believes ((LRRH boy) & (W fox)))
[947] [949]
==> (John believes (wolf-eat-lrrh before-1 wolf-eat-gm))
[951]
```

```
==> trace spec-res spec-eval spec-fact qa-time
(SPEC-RESOLVABLE SPEC-EVALUATION SPEC-FACTORABLE QA-TIME)

==> ?(wolf-eat-lrrh before wolf-eat-gm)
   Specialists: Evaluating (WOLF-EAT-LRRH BEFORE WOLF-EAT-GM)
   Used 3.36 seconds CPU time  3.4 seconds real time
NO
==> ?(John believes (wolf-eat-lrrh before wolf-eat-gm))
   Specialists: Evaluating (JOHN BELIEVES (WOLF-EAT-LRRH BEFORE WOLF-EAT-GM))
   Used .58 seconds CPU time  .58 seconds real time
YES
==> untrace spec-res spec-eval spec-fact
(SPEC-RESOLVABLE SPEC-EVALUATION SPEC-FACTORABLE)

; other tests
==> trace number-entry number-prop
(NUMBER-ENTRY NUMBER-PROPAGATION)
==> (n1_integer lt n2_real)
   Number specialist: Entering N1 LT N2
[954]
==> (n1 gt '5.5)
   Number specialist: Entering N1 GT (QUOTE 5.5)
   Number specialist: Setting minimum of N1 to 6
   Number specialist: Propagating minimum 6.0 to N2
[955]
==> (n2 lt= '7)
   Number specialist: Entering N2 LT= (QUOTE 7)
   Number specialist: Setting maximum of N2 to 7.0
   Number specialist: Propagating maximum 7 to N1
[956]
==> untrace number-entry number-prop
(NUMBER-ENTRY NUMBER-PROPAGATION)

==> trace number-fac number-res
(NUMBER-FACTORABLE NUMBER-RESOLVABLE)
==> ?(a x (x lt n1) => (x lt n2))
   Number Specialist: Trying to factor (U-VAR-1 LT N2) and (~ U-VAR-1 LT N1)
   Number Specialist: Factored to (N1 LT N2)
   Used .86 seconds CPU time  .92 seconds real time
YES

==> untrace number-fac number-res
(NUMBER-FACTORABLE NUMBER-RESOLVABLE)

==> (lrrh-people_set equal (set-of LRRH GM WOODCUTTER))
[960]
==> (lrrh-chars_set equal (set-of lrrh-people W))
[964]
==> (lrrh-entities_set equal (set-of lrrh-chars BAS))
[968]
```

```
==> trace set-eval number-eval
(SET-EVALUATION NUMBER-EVALUATION)
==> ?(LRRH member-of-0 lrrh-entities)
   Set Specialist: Evaluating LRRH MEMBER-OF-0 LRRH-ENTITIES
   Set Specialist: Evaluated LRRH MEMBER-OF-0 LRRH-ENTITIES to YES
   Used .12 seconds CPU time  .12 seconds real time
YES
==> ?(lrrh-chars member-of lrrh-entities)
   Set Specialist: Evaluating LRRH-CHARS MEMBER-OF LRRH-ENTITIES
   Set Specialist: Evaluated LRRH-CHARS MEMBER-OF LRRH-ENTITIES to YES
   Used .08 seconds CPU time  .08 seconds real time
YES

==> set assert major
==> (a x_set ((number-members-0 x) gt (number-members x)) => (x complex-set))
[977]
==> set assert nomajor

==> trace qa-successes
(QA-SUCCESSES).
==> ?(lrrh-entities complex-set)
Entering disproof clauses:
(LRRH-ENTITIES COMPLEX-SET) [978] (depth 1)

Entering proof clauses:
(~ LRRH-ENTITIES COMPLEX-SET) [979] (depth 1)

  Proof access for clause (~ LRRH-ENTITIES COMPLEX-SET) [979]
     {Major imp for lit [979] at COMPLEX-SET->TP.MAJOR-IMP}
     Ranking: 9300  Depth: 1
    Yielded 1 resolving agenda items after checking
    against 1 propositions.

  Resolved (~ LRRH-ENTITIES COMPLEX-SET) in the proof clause
    (~ LRRH-ENTITIES COMPLEX-SET)
    against (SET-VAR-1 COMPLEX-SET) in
    ((~ (NUMBER-MEMBERS-0 SET-VAR-1) GT (NUMBER-MEMBERS SET-VAR-1))
       | (SET-VAR-1 COMPLEX-SET))
    yielding ...

(~ (NUMBER-MEMBERS-0 LRRH-ENTITIES) GT (NUMBER-MEMBERS LRRH-ENTITIES)) [983] (depth 2)
  Number specialist: Evaluating (QUOTE 5) GT (QUOTE 2)
  Number Specialist: Evaluated (QUOTE 5) GT (QUOTE 2) to YES
  ((NUMBER-MEMBERS-0 LRRH-ENTITIES) GT (NUMBER-MEMBERS LRRH-ENTITIES))
    evaluated to YES.
  The clause evaluated to NO.
  Used .48 seconds CPU time  .5 seconds real time
YES
```